Tampereen yliopisto

Teemu Kuikka

REAALIAIKAINEN TYÖKONEEN SEURANTA RATATYÖKONEIDEN SUORITEKIRJAUSJÄRJESTELMÄSSÄ

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Kesäkuu 2019

TIIVISTELMÄ

Teemu Kuikka: Reaaliaikainen työkoneen seuranta ratatyökoneiden suoritekirjausjärjestelmässä
Diplomityö
Tampereen yliopisto
Tietotekniikka
Kesäkuu 2019

Tämä työ käsittelee työvuoronkirjausprosessin tiedonkulun reaaliaikaistamista web-pohjaisessa ratatyökoneiden suoritekirjausjärjestelmässä. Suoritekirjausjärjestelmä koostuu backend-palvelimesta ja verkkoselaimella käytettävistä kirjaus- ja raportointisovelluksista. Järjestelmän kirjaussovellus on tarkoitettu ratatyökoneiden miehistön käyttöön ja sitä käytetään ratatyön yhteydessä työvuorojen tapahtumien kirjaamiseen. Raportointisovellusta käyttävät esimiehet ja sen avulla voidaan esimerkiksi seurata valmiista työvuoroista johdettuja raportteja ja tilastoja, tarkastaa ja hyväksyä tehtyjä työvuoroja ja luoda työvuorosuunnitelmia.

Ennen työn toteuttamista työvuoron kirjausprosessi toimi siten, että konemies kirjasi työvuoronsa tapahtumat kirjaussovellukseen, jonka välimuistissa työvuoroa säilytettiin, kunnes se oli kokonaan valmis. Kun työvuoro oli valmis, se lähetettiin kirjaussovelluksesta järjestelmän backend-palvelimelle, jolloin työvuoro ja sen tiedot tulivat tarkasteltaviksi myös raportointisovelluksessa. Tämän työn pääasiallinen kehitystavoite oli reaaliaikaistaa tätä tiedonkulkua siten, että tietoa työvuoroista saataisiin raportointisovellukseen jo silloin, kun työvuoroa tehdään, eikä vasta sen valmistuttua.

Järjestelmän raportointisovellukseen haluttiin toteuttaa käyttöliittymä, jonka kautta meneillään olevien työvuorojen tapahtumia voidaan seurata työkonekohtaisesti aikajanalla sekä kronologisena listauksena. Tätä varten tarvittiin mekanismi, jolla kirjaussovellus lähettää tapahtumapäivityksiä palvelimelle työvuoron edetessä ja jolla palvelin voi lähettää päivitykset eteenpäin raportointisovellukselle. Tämä ei onnistu yksistään HTTP-protokollaa käyttämällä. Työn tekninen tavoite oli siis löytää palvelimelta raportointisovellukselle suuntautuvalle reaaliaikaiselle tiedonkululle toteutustapa, joka on verkkoyhteyden datankäytön suhteen tehokas, helposti toteutettavissa olemassa olevaan järjestelmään ja joka potentiaalisesti tarjoaisi järjestelmälle uusia jatkokehitysmahdollisuuksia, kun pohjatoteutus on olemassa.

Reaaliaikaisen viestinnän toteutukselle vertailtiin kolmea erilaista menetelmää: HTTP-protokollan soveltamista, WebSocket-protokollaa ja Server Sent Events -teknologiaa. Vertailu tehtiin teknisen tavoitteen perusteella ja toteutustavaksi valittiin WebSocket-protokolla. Server Sent Events -teknologia katsottiin myös hyväksi vaihtoehdoksi, mutta WebSocket-protokollan katsottiin tarjoavan parempia jatkokehitysmahdollisuuksia.

Järjestelmän backend-palvelimelle ja kirjaus- ja raportointisovelluksille toteutettiin tuki WebSocket-protokollalle olemassa olevien kirjastojen avulla. Työvuorojen tapahtumapäivitysten välittämiselle määriteltiin viestintäprotokolla, jolle toteutettiin myös tuki järjestelmän komponentteihin. Viestintäprotokolla suunniteltiin siten, että se mahdollistaa helpon laajennettavuuden ja siten jatkokehitettävyyden uusilla WebSocket-protokolla hyödyntävillä ominaisuuksilla. Raportointisovellukseen luotiin toteutustavoitteiden mukainen seurantakäyttöliittymä.

Toteutusta arvioidessa todettiin, että työn kehitystavoitteet on onnistuneesti saavutettu. WebSocket-protokolla todettiin toimivaksi menetelmäksi reaaliaikaisen kaksisuuntaisen kommunikaation toteutukselle ja sen havaittiin tarjoavan järjestelmälle hyviä jatkokehitysmahdollisuuksia.

Avainsanat: HTTP, WebSocket, Server Sent Events, web

ABSTRACT

Teemu Kuikka: Real-time work train tracking in a work train performance logging system
Master of Science Thesis
Tampere University
Information Technology
June 2019

This thesis concerns adding a real-time element to the information flow of a web based work train performance logging system's work shift logging process. The performance logging system consists of a backend server, as well as a logging application and a reporting application that are used with an internet browser. The logging application is intended to be used by the work trains' crew and it is used to log the events of work shifts. The reporting application is used by superiors and it can be used e.g. to view reports and statistics derived from the events of finished work shifts, inspect and accept logged work shifts and create plans for future work shifts.

Prior to the implementation of this thesis, the work shift logging process worked so that the work machine's crew logged the events of the work shift into the logging application and the information of the unfinished work shift was kept in local cache until it was complete. When the work shift was finished, it and all of its details were sent to the backend server. After that the information of the work shift was accessible in the reporting application. The main development goal of this thesis was to make the information of a work shift's events available to the reporting application in real-time as soon as they were logged in the logging application, instead of only after the work shift was complete and sent to the backend server.

The desire was to implement a user interface into the reporting application, where the events of ongoing work shifts could be monitored per machine on a timeline as well as in a chronological list. To achieve this, a mechanism was needed to allow delivering event updates from the logging application to the backend server that then delivered them to the reporting application. This could not be done using HTTP alone. The technical development goal of this thesis was to find a solution to deliver data in real-time from the server to the reporting application. This solution needed to be efficient in regards to data consumption, be easily implementable into the existing system and potentially offer possibilities for further development taking advantage of the base implementation.

Three different methods were compared for the implementation of real-time communication: unconventional application of the HTTP protocol, WebSocket protocol and Server Sent Events. The comparison was made according to the technical goal of this thesis and WebSocket was chosen as the implementation method. Server Sent Events was also considered as a potential method but WebSocket seemed to offer better possibilities for further development.

Support for WebSocket protocol was implemented into the backend server and the logging and reporting applications using existing libraries. A communication protocol was specified for the delivery of the work shift event updates and support for it was implemented into the system's components as well. The communication protocol was designed in a way that allows for easy expansion and thus further development possibilities with new features utilizing the WebSocket protocol. A work shift tracking user interface was implemented into the reporting application based on the development goals of this thesis.

Upon examining the finished implementation of this thesis it was found that the development goals were successfully achieved. WebSocket was deemed a suitable method to implement real-time two-way communication and it was found to offer good possibilities for further development.

Keywords: HTTP, WebSocket, Server Sent Events, web

ALKUSANAT

Suunniteltua kauemmin siinä kesti, mutta nyt diplomityö on valmis ja sen myötä yksi vaihe elämässä. Haluan kiittää työnantajaani ja asiakastani työn toteuttamisen mahdollistamisesta, työkavereitani ja ystäviäni arvokkaasta vertaistuesta ja ideoista, läheisiäni kannustamisesta sekä ohjaajaani Kari Systää kaikesta matkan varrella saadusta palautteesta ja mielenkiinnosta työtä kohtaan.

Tampereella, 2.5.2019

Teemu Kuikka

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

API	Application Programming Interface, ohjelmointirajapinta
DOM	Document Object Model, verkkosivun elementtien sisäkkäistä rakennetta kuvaava puumalli
HTTP	Hypertext Transfer Protocol, hypertekstin siirtoprotokolla
JSON	Java-Script Object Notation, dataobjektien rakenteen määrittelykieli
OSI-malli	Open Systems Interconnect -malli, verkkoprotokollien kerrosmalli
SVG	Scalable Vector Graphics, vektorikuvien kuvauskieli
TCP	Transmission Control Protocol, tietoliikenneprotokolla
URI	Uniform Resource Identifier, merkkijono, joka yksilöi tietyn resurssin
URL	Uniform Resource Locator, verkko-osoite, referenssi verkossa sijaitsevaan resurssiin

1. JOHDANTO

Ohjelmistoyritys Bitwise toteuttaa rautatiealalla toimivalle asiakasyritykselleen web-pohjaista ratatyökoneiden suoritekirjausjärjestelmää. Tämän järjestelmän avulla ratatyökoneiden kuljettajat voivat kirjata ylös työvuoronsa tapahtumia sen edetessä ja esimiehet voivat seurata työvuoroista johdettuja tilastoja sekä työvuororaportteja.

Työvuorojen tapahtumat kirjataan järjestelmän verkkoselaimella käytettävällä kirjaussovelluksella dynaamisesti työvuoron edetessä, mutta ne raportoidaan järjestelmän muille osille vasta työvuoron päätyttyä. Kirjaus tapahtuu mobiilin datayhteyden välityksellä. Työvuoroja on osittain mahdollista kirjata myös ilman datayhteyttä säilyttäen lähettämättömät kirjaukset selaimen välimuistissa, kunnes ne voidaan lähettää.

Näillä lähtökohdilla kirjausjärjestelmän tiedonkulkua haluttiin kehittää reaaliaikaisemmaksi ja informatiivisemmaksi, jotta työvuorojen tapahtumia voitaisiin seurata samaan aikaan, kun suoritekirjauksia tehdään. Tämä työ käsittelee tarvittavan reaaliaikaisuuden ja siihen liittyvien ominaisuuksien toteuttamista järjestelmään.

Reaaliaikaisuuden tuomista osaksi järjestelmää lähestyttiin aluksi määrittelemällä ominaisuudet, jotka reaaliaikaisuudella haluttiin mahdollistaa sekä tutkimalla erilaisia vaihtoehtoisia tekniikoita reaaliaikaisen kommunikoinnin toteutukselle palvelimen ja asiakassovellusten välille. Toteutuksessa olennaista oli myös reaaliaikaisten viestien hallinta palvelimella ja seurantakäyttöliittymän suunnittelu ja toteutus.

Tässä työssä käydään läpi ominaisuuksien suunnittelu- ja toteutusprosessi alkaen järjestelmän esittelystä ja päättyen mahdollisten jatkokehitysvaihtoehtojen pohtimiseen. Luvussa 2 käydään läpi kirjausjärjestelmän arkkitehtuuria ja ominaisuuksia. Luvussa 3 käydään läpi järjestelmän kehitystavoitteet. Luvussa 4 esitellään järjestelmässä käytetyt teknologiat. Luvussa 5 kuvataan kehitettävien ominaisuuksien suunnittelu- ja toteutusprosessia. Luvussa 6 arvioidaan toteutuksen laatua ja toimivuutta. Luvussa 7 pohditaan reaaliaikaisuuden jatkokehitysmahdollisuuksia järjestelmässä. Luvussa 8 kootaan yhteenveto työstä.

2. JÄRJESTELMÄN KUVAUS

Järjestelmä on web-pohjainen suoritekirjausjärjestelmä, joka on toteutettu asiakkaan ratatyökoneiden koneliiketoiminnan tarpeisiin. Järjestelmää käytetään verkkoselaimella ja se jakautuu käyttäjän näkökulmasta erillisiin kirjaus- ja raportointisovelluksiin.

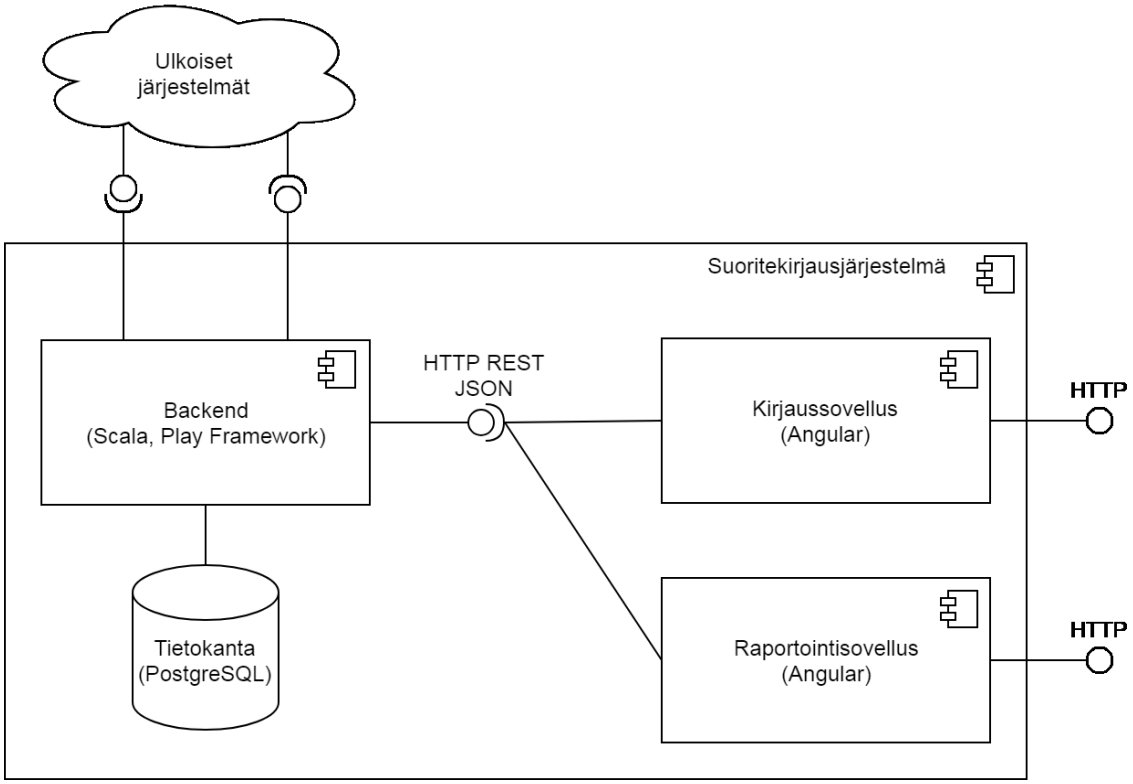
Kirjaussovellus on tarkoitettu työkoneiden miehistön käyttöön. Se on ensisijaisesti suunniteltu käytettäväksi Panasonic Toughpad -tabletilla, mutta sen käyttö onnistuu myös millä tahansa muulla laitteella, jossa on verkkoselain. Sen käyttöliittymä tukee myös esimerkiksi erilaisia mobiililaitteita, kuten matkapuhelimia.

Työkoneiden tehtävien luonteen vuoksi kirjaussovellusta käytetään tyypillisesti mobiiliverkon välityksellä. Joskus työtä voidaan tehdä paikoissa, joissa verkon kuuluvuus on huono, joten kirjaussovelluksessa on osittainen tuki offline-käytölle. Sovelluksen avaaminen vaatii toimivan internetyhteyden, mutta esimerkiksi keskeneräiset ja valmiit kirjaukset voidaan pitää tallessa selaimen välimuistissa, kunnes ne voidaan lähettää järjestelmän muille osille.

Raportointisovelluksen käyttäjiin kuuluvat ensisijaisesti työkoneryhmien esimiehet, järjestelmän ylläpitäjät ja asiakkaalta ratatöitä tilaavat tahot. Myös työkoneiden miehistö voi käyttää sovellusta erilaisten tilastojen seuraamiseen. Raportointisovellus on tarkoitettu käytettäväksi työpöytäselaimilla eikä sille näin ollen ole toteutettu erillistä mobiilikäyttöliittymää.

2.1 Järjestelmän arkkitehtuuri ja ympäristöt

Suoritekirjausjärjestelmä koostuu kolmesta pääkomponentista: selainkäyttöisistä kirjaus- ja raportointisovelluksista sekä backend-palvelimesta. Kirjaus- ja raportointisovellukset on toteutettu Angular -alustalla ja backend-palvelin Scala -ohjelmointikielellä käyttäen Play Framework -sovelluskehystä. Backend-palvelin on myös yhteydessä PostgreSQL -tietokantaan. Järjestelmän toteutusteknologioista kerrotaan laajemmin luvussa 4. Tämän lisäksi järjestelmällä on integraatioita ulkoisiin järjestelmiin, joista kerrotaan luvussa 3.3.1. Järjestelmän korkean tason arkkitehtuuri on esitetty kuvassa 1.

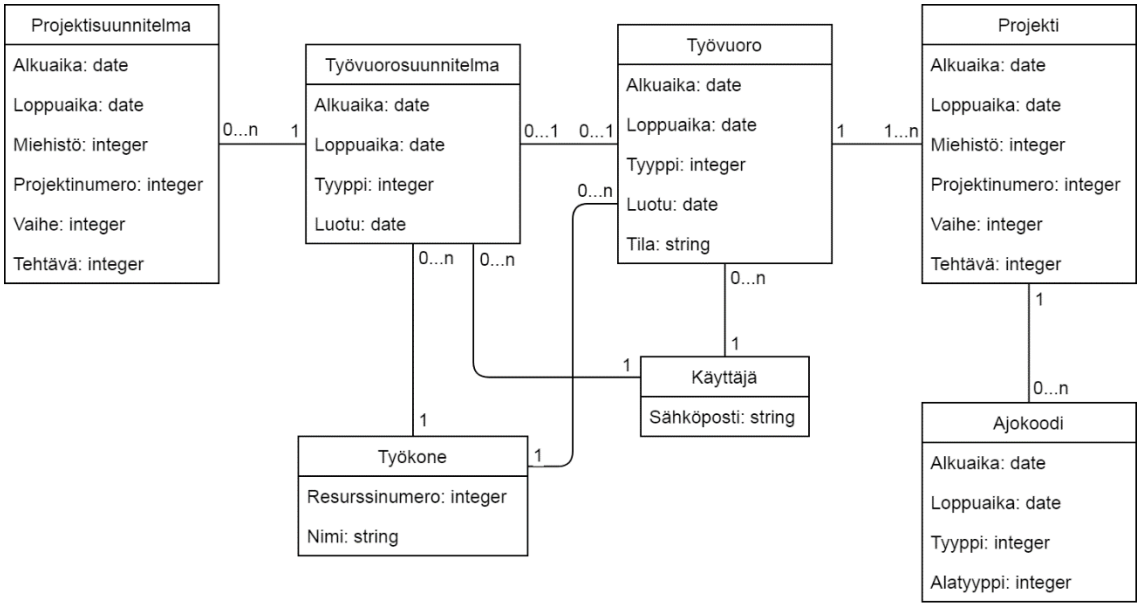


Kuva 1. *Suoritekirjausjärjestelmän arkkitehtuuri*

Suoritekirjausjärjestelmästä on olemassa erikseen testiympäristö ja tuotantoympäristö. Molempia ympäristöjä ajetaan virtuaalikoneilla. Testiympäristössä on ajossa järjestelmän versio, jossa on mukana uusia kehitettyjä ominaisuuksia, joita asiakas voi testata ja tarvittaessa ominaisuuksia jatkokehitetään ennen tuotantoympäristöön julkaisemista. Tuotantoympäristössä on ajossa järjestelmän uusin toimivaksi todettu ja hyväksytty tuotantoversio.

2.2 Työvuoron käsite

Työvuoron käsite on suoritekirjausjärjestelmän kaikkien komponenttien ja niiden toiminnallisuuksien kannalta keskeinen. Työvuorot kirjataan kirjaussovelluksessa ja niistä johdettua dataa, kuten esimerkiksi työraportteja, tarkastellaan raportointisovelluksessa. Työvuoroista muodostetut työraportit on asetettu saataville myös asiakkaan muille järjestelmille ulkoisen rajapinnan välityksellä. Työvuoron sekä siihen liittyvien käsitteiden tietomallia on kuvattu kuvan 2 kaaviossa.



Kuva 2. *Työvuoron ja siihen liittyvien osien UML-kaavio*

Kuvan 2 kaavio vastaa karkeasti järjestelmän sisältämää tietomallia työvuorosta ja siihen keskeisesti liittyvistä muista käsitteistä. Suurin osa työvuoron käsitteen kannalta merkityksettömistä tiedoista on jätetty selkeyden vuoksi kaaviosta pois. Järjestelmä sisältää todellisuudessa esimerkiksi työkoneista paljon muutakin tietoa, kuin resurssinumeron ja nimen.

Tämän työn kannalta tärkeimpänä seikkana kaaviosta on syytä huomata, että työvuoro koostuu yhdestä tai useammasta projektista ja projektit taas voivat koostua ajokoodeista. Osa järjestelmän työkoneista ei käytä ajokoodeja, joten projekti voi näissä tapauksissa olla sisältämättä ajokoodeja.

Ajokoodilla tarkoitetaan projektin osaa, jonka aikana on tehty tietynlaista työtä. Ajokoodin tyyppiä kuvataan sille määrätyllä numerolla. Joitain työtyyppejä voidaan täsmentää myös alatyypillä. Järjestelmän erilaiset työtyypit ovat

- tehollinen työ, alatyypein
    a. vaihdetyö,
    b. raidetyö,
- Siirto,
- Liikennehaitta,
- Työnjärjestelytauko,
- Korjaus ja muu tauko

Ajokoodeilla on alku- ja loppuaika ja oikeanmuotoisessa työvuorossa saman projektin sisällä olevat ajokoodit muodostavat alku- ja loppuajoillaan katkeamattoman ketjun projektin alkuajan ja loppuajan välillä.

Projektit sisältyvät työvuoroon samaan tapaan, kuin ajokoodit projektiin. Jos työvuorossa on useampi projekti, niiden on oltava aikojensa puitteissa peräkkäisiä katkeamattomana ketjuna sekä niiden aikojen on kuuluttava työvuoron alku- ja loppuaikojen väliin. Projektin tarkoitus ja sisältö tunnistetaan sen projektinumeron, vaiheen ja tehtävän avulla. Nämä luvut kirjataan projekteille niiden alkaessa ja niiden avulla voidaan laskea tehdystä työstä erilaisia tilastoja. Niitä hyödyntävät myös asiakkaan muut järjestelmät.

2.3 Järjestelmän ominaisuudet

Suoritekirjausjärjestelmän pääasiallinen käyttötarkoitus on työkoneiden työvuorosuoritteiden kirjaaminen niiden tapahtuessa sekä kirjanpito tehdyistä työvuoroista asiakkaan muita järjestelmiä ja sen työkonetoiminnan kehittämistä varten. Järjestelmä sisältää tämän ohella myös muita ominaisuuksia, joita tässä luvussa käydään läpi.

2.3.1 Ajastetut tehtävät ja integraatiot

Suoritekirjausjärjestelmä käyttää ulkoisia rajapintoja työvuoron tietojen kirjauksenaikaiseen täydentämiseen. Järjestelmä käyttää Enrico –juhlapyhärajapintaa [1] Suomen juhlapyhien ajankohtien tiedustelemiseen. Juhlapyhätietojen avulla työvuoro voidaan kirjauksen aikana automaattisesti merkitä joko arki- tai pyhäpäivänä tapahtuneeksi. Juhlapyhätiedot haetaan automaattisesti kerran viikossa yöaikaan ja tallennetaan tietokantaan.

Lisäksi järjestelmä käyttää Liikenneviraston Infa-API –rajapintaa [2] Suomen rautatieliikennepaikkojen ja tilirataosien nimien hakemiseen. Rautatieliikennepaikkoja ja tilirataosia käytetään, kun kirjaussovelluksessa kirjataan paikka, jossa työtä tehdään. Rautatieliikennepaikat ja tilirataosat voivat muuttua ajoittain, joten niitä haetaan kerran vuorokaudessa yöaikaan ja ne tallennetaan tietokantaan.

Suoritekirjausjärjestelmä tarjoaa asiakkaan toiminnanohjausjärjestelmälle ulkoisen REST-rajapinnan, jonka kautta työraporttimuotoisia työvuorotietoja voidaan hakea. Työvuorotietoja voidaan hakea vapaasti määritellyllä aikavälillä. Asiakkaan toiminnanohjausjärjestelmä hakee työvuorotietoja tästä rajapinnasta säännöllisin väliajoin.

Ajastettujen rajapintaintegraatioiden lisäksi suoritekirjausjärjestelmä lähettää joitain ajastettuja tai muin ehdoin lähetettäviä sähköpostiviestejä järjestelmän pääkäyttäjille. Tällaisia ovat esimerkiksi ajastetut koosteraportit koneiden käyttötunneista tai koneen

kirjattujen käyttötuntien perusteella aktivoituvat lähestyvästä huoltotarpeesta ilmoittavat viestit.

2.3.2 Työraportit

Raportointisovelluksessa voidaan seurata kirjatuista työvuoroista muodostettuja raportteja taulukkomuodossa. Raportit johdetaan työvuoroista siten, että ajokoodeja käyttävien koneiden tekemien työvuorojen osalta jokainen ajokoodi saa oman rivinsä ja koneiden, jotka eivät käytä ajokoodeja, osalta jokainen projekti saa oman rivinsä. Ajokoodeja käyttävien koneiden työvuoroihin lisätään myös tietyissä tapauksissa täydennysrivi, jolla työvuoron kesto täydennetään tietyin ehdoin säännöllisen mittaiseksi sen ollessa vajaa.

Yksittäinen työraportti sisältää erilaisia tietoja, kuten esimerkiksi työvuoron alku- ja loppuajat, itse raportin käsittelemän työn osan alku- ja loppuajat, projektin tunnisteen eli projektinumeron, työvaiheen ja tehtävän, tehdyn työn tyypin sekä tehollisella työllä toteutuneet suoritteet. Työraporttimuotoista dataa käytetään kirjanpitoon asiakkaan muissa järjestelmissä eli se on pääasiallinen muoto, jossa tehtyä työtä lopullisesti seurataan. Työraporttitaulukon voi myös ladata järjestelmästä Excel-muodossa.

Työraporttinäkymässä voidaan tarkastella taulukkomuodossa myös muita konemiesten kirjauksen aikana lähettämiä oheistietoja. Tällaisia tietoja ovat viestit, tankkausilmoitukset, koneiden tarkistuslomakkeet, konetuntikirjaukset, työvuorojen palautteet sekä kuvat.

Suurinta osaa raportointisovelluksen näkymistä voidaan suodattaa muutaman erilaisen parametrin mukaan. Näytettävää dataa voidaan rajata päivämäärävälillä, projektinumerolla, työvaiheella ja tehtävänumerolla sekä koneilla. Esimerkiksi raporttitaulukkoon voidaan rajata näkyväksi vain tietyn koneen tietyllä projektinumerolla tehdyistä työvuoroista muodostetut raportit.

2.3.3 Työvuorojen suunnittelu

Raportointisovellus sisältää työvuorojen suunnitteluominaisuuden, jonka avulla työkoneille voidaan määrittää työvuoroja etukäteen. Työvuorosuunnitelmaan voidaan määrittää työvuoron päivämäärä, alku- ja loppuajankohdat ja esimerkiksi työvuoroon liittyvän yhteyshenkilön yhteystiedot. Lisäksi suunnitelmalle voidaan lisätä projekteja, joille voidaan määrittää valmiiksi projektinumero, työvaihe ja tehtävä, kesto, työntekijöiden lukumäärä sekä mahdolliset työkoneeseen kytkettävät vaunut ja lisälaitteet. Työvuorosuunnitelmia voidaan lisätä järjestelmään joko järjestelmässä olevan lomakkeen kautta tai lataamalla järjestelmään Excel-tiedosto, jossa työvuorosuunnitelmat on määritetty oikeanlaisella rakenteella.

Luotuja työvuorosuunnitelmia voidaan tarkastella joko listausnäkymässä tai aikajanana gantt-kaaviossa. Gantt-kaavio on visuaalinen esitystapa, jossa on vaakasuuntainen aikajana, johon nykyhetki on merkitty sekä sen alla riveittäin esimerkiksi tehtäviä, joiden kestosta piirretään palkki oikeaan kohtaan aikajanalla. Suunnittelunäkymän gantt-kaaviossa riveillä ovat koneet, joille suunnitelmia on tehty ja niille piirrettävät palkit kuvaavat työvuorosuunnitelmien ajankohtia.

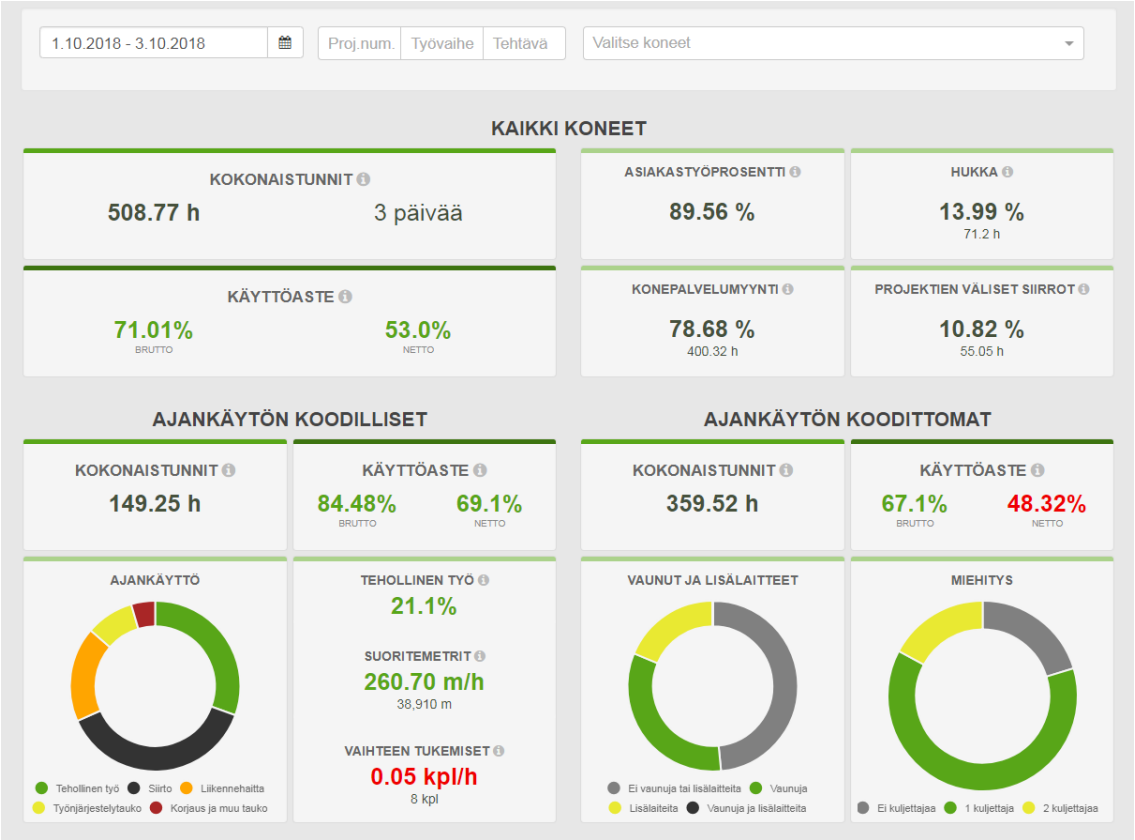
2.3.4 Työvuorojen hyväksyntä

Kun konemies päättää työvuoronsa ja lähettää kirjauksensa järjestelmään, se tulee näkyviin raportointisovelluksen hyväksyntänäkymään, josta esimiehet voivat niitä tarkastella. Tässä näkymässä työvuorot näytetään listana ja yksittäinen työvuoro voidaan laajentaa suuremmaksi ja tarkastella kaikkia työvuoron aikana kirjattuja tietoja.

Työvuoroja voidaan verrata suunnitelmiin, joiden pohjalta ne on toteutettu sekä tarvittaessa niiden kaikkia tietoja voidaan muokata. Lopuksi työvuorot voidaan joko hylätä tai hyväksyä alkuperäisessä tai muokatussa tilassa. Kun työvuoro on hyväksytty, se tulee näkyviin työraporttien listaukseen.

2.3.5 Dashboard

Raportointisovelluksessa on dashboard-näkymä, josta voidaan tarkastella kootusti erilaisia koneisiin ja niiden käyttöön liittyviä tilastoja. Käyttämällä raportointisovelluksen rajaustoimintoja näitä tilastoja voidaan tarkkailla hyvin monipuolisesti esimerkiksi lyhyellä tai pitkällä aikavälillä tai ainoastaan tietyn tyyppisillä koneilla. Dashboardin mittarit on jaettu kolmeen ryhmään, jotka ovat kaikki koneet, ajankäytön koodilliset eli koneet, jotka käyttävät ajokoodeja ja ajankäytön koodittomat eli koneet, jotka eivät käytä ajokoodeja. Dashboard-näkymän ulkoasu on esitetty kuvassa 3.



Kuva 3. *Raportointisovelluksen dashboard-näkymä*

Järjestelmään on myös kehitetty erillinen dashboard -osio konemiehille. Konemiesten dashboardissa konemies voi nähdä oman koneensa kokonaistunnit ja käyttöasteet. Sen kautta voidaan myös välittää konemiehille ilmoituksia ja näyttää järjestelmään erikseen ladattuja kuvamuodossa olevia mittareita.

2.3.6 Kirjaussovellus ja kirjausprosessin kulku

Kirjaussovelluksen tärkein ominaisuus on työvuoron tapahtumien kirjaus sen edetessä ja työvuoron lähettäminen palvelimelle. Kirjaussovellus kirjaa automaattisesti työvuoron osakokonaisuuksien alku- ja loppuajat, kun työvuoron tilaa muutetaan sen edetessä. Työvuoroa ja sen osakokonaisuuksia kellonaikoineen voidaan kuitenkin tarvittaessa muokata vielä työvuoron päätyttyä ennen sen lähettämistä.

Kirjaussovellus pitää sitä käytettävällä laitteella kesken tai lähettämättä jääneiden työvuorojen kirjauksen tilan tallessa selaimen välimuistissa. Sovelluksella voidaan nähdä listaus lähettämättömistä työvuoroista ja tämän listausnäkymän kautta muistissa olevat valmiit työvuorot voidaan joko lähettää palvelimelle tai poistaa välimuistista.

Työvuoron tapahtumien kirjaamisen ohella kirjaussovelluksella voidaan milloin vain kirjauksen aikana suorittaa tarvittaessa myös erilaisia lisätoimintoja. Tällaisia toimintoja ovat valokuvan ottaminen ja lähettäminen palvelimelle, tankkausilmoituksen lähettäminen, vikaviestin lähettäminen ja muun esimuotoillun tai vapaamuotoisen viestin lähettäminen.

Kirjausprosessi alkaa työkoneen valinnasta. Työkoneen valintanäkymässä valitaan alasvetovalikosta työvuoron suorittava työkone sekä kirjataan koneen käyntituntimittarin ilmoittama käyntituntilukema ennen työvuoron aloittamista. Kun kone on valittu ja käyntitunnit kirjattu, voidaan siirtyä eteenpäin. Jos välimuistissa on olemassa työkoneelle keskeneräinen työvuoron kirjaus, sovellus ilmoittaa siitä eteenpäin siirryttäessä ja kysyy, haluaako käyttäjä jatkaa vanhaa työvuoroa vai aloittaa uuden. Vanhaa työvuoroa jatkettaessa sovellus siirtyy suoraan projektinvalintanäkymään. Jos aloitetaan uusi työvuoro, siirrytään tarkastuslomakenäkymään.

Tarkastuslomakenäkymässä esitetään työvuorolle valitun työkoneen ajoonlähtötarkastuksen tehtävälista. Tässä vaiheessa konemies suorittaa tehtävälistassa mainitut toimenpiteet ja lisää halutessaan vapaamuotoisen huomautuksen tarkastukseen liittyen. Lopuksi tarkastus kuitataan tehdyksi ja siirrytään kirjausprosessissa eteenpäin.

Jos koneelle on tehty raportointisovelluksessa työvuorosuunnitelmia, seuraavaksi avautuu suunnitelman valintanäkymä, jossa voidaan valita toteutettava suunnitelma tai edetä ilman suunnitelmaa. Jos suunnitelmia ei ole, avautuu työvuoron suunnitellun ajankohdan valintanäkymä, jossa kirjataan työvuoron suunnitellun alkamisen ja päättymisen kellonajat sekä valitaan, toteutetaanko työvuoro arki- vai pyhäpäivänä. Jos suunnitelma valittiin aiemmin, ajankohta ja pyhäpäivätilanne ovat esitäytetyt suunnitelman mukaisesti. Normaalisti pyhäpäivät tunnistetaan automaattisesti, mutta tarvittaessa valinta voidaan tehdä myös manuaalisesti. Kun työvuoron ajankohta on asetettu, voidaan siirtyä eteenpäin projektinvalintanäkymään.

Projektinvalintanäkymässä valitaan aloitettavan projektin projektinumero, työvaihe ja tehtävänumero. Luvut voidaan syöttää kenttiin manuaalisesti tai käyttää tarjottuja lukuja. Sovellus tarjoaa valmiina lukuina sellaisia lukuja, joita työkoneella on aiemmin käytetty, koneelle määriteltyjä vakioarvoja sekä suunnitelmassa määriteltyjä lukuja, jos sellaisia on. Projektin valintanäkymän jälkeen kirjausprosessi jatkuu eri tavalla ajankäytön koodillisten ja ajankäytön koodittomien koneiden välillä.

Ajankäytön koodillisilla koneilla kirjaussovellus siirtyy projektinvalintanäkymästä ajankäytön hallintanäkymään. Tämä näkymä sisältää painikkeet kaikille järjestelmän eri ajokoodeille. Kun jotain ajokoodia vastaavaa työtä aletaan tekemään, painetaan ajokoodia

vastaavaa painiketta, jonka jälkeen sovellus alkaa mittaamaan ajokoodille käytettyä aikaa. Kun aletaan tekemään toisen tyyppistä työtä, painetaan uutta painiketta, jolloin sovellus tallentaa aiempaan ajokoodiin kulutetun ajan ja alkaa laskea aikaa uudelle ajokoodille. Jos aloitetaan tehollinen työ –ajokoodi, sovellus kysyy työn alatyyppiä, eli tehdäänkö raide- vai vaihdetyötä sekä sijaintia, jossa työtä tehdään. Työn sijaintitietoa varten kysytään tilirataosaa, liikennepaikkaa, raidetta sekä raiteen ratakilometrisijaintia. Kun tehollinen työ vaihdetaan toiseen ajokoodiin, kysytään vielä uutta ratakilometrisijaintia, jonka avulla lasketaan, kuinka pitkällä matkalla työtä on tehty. Vaihdetyössä kysytään lisäksi myös tuettujen vaihteiden lukumäärää. Näkymästä löytyvät painikkeet myös uuden projektin aloittamiselle ja työvuoron lopettamiselle. Jos näkymässä aloitetaan uusi projekti, nykyinen projekti tallennetaan ja siirrytään takaisin projektinvalintanäkymään. Työvuoron lopetuspainiketta painettaessa koko työvuoro tallennetaan ja siirrytään palautenäkymään.

Ajankäytön koodittomilla koneilla projektinvalintanäkymästä siirrytään miehitysnäkymään, jossa voidaan valita koneen mukana työtä tekevien konemiesten lukumäärä. Tästä siirrytään vaunujen ja lisälaitteiden valintanäkymään, jossa valitaan koneessa kiinni mahdollisesti olevat vaunut ja mukana olevat lisälaitteet, joita työssä tarvitaan. Kun vaunut ja lisälaitteet on valittu, siirrytään työnäkymään. Työnäkymässä sovellus laskee työhön käytettyä aikaa, kuten ajankäytön koodillisten koneiden ajankäytön hallintanäkymässä, mutta siinä ei valita erikseen ajokoodeja, vaan aikaa lasketaan koko projektille. Tässä näkymässä on myös mahdollista vaihtaa uuteen projektiin, kuten ajankäytön koodillisten koneiden ajankäytön hallintanäkymässä. Kun tässä näkymässä päätetään työvuoro, siirrytään palautenäkymään.

Myös palautenäkymä on hieman erilainen ajankäytön koodillisten ja koodittomien koneiden välillä, mutta sen toimintaperiaate on molemmilla sama. Näkymässä kysytään erilaisia työvuoroon liittyviä kysymyksiä, joihin konemies vastaa joko kyllä tai ei. Lisäksi näkymä sisältää vapaamuotoisesti täytettävän yleisen palautteen tekstikentän työvuoroon liittyen. Kun palautenäkymästä jatketaan eteenpäin, siirrytään työvuoron yhteenvetonäkymään.

Työvuoron yhteenvetonäkymässä listataan koko kirjattu työvuoro ja sen kaikki yksityiskohdat. Ajankäytön koodillisilla koneilla listataan työvuoroon sisältyvät projektit sekä projektien sisällä ajokoodit. Ajankäytön koodittomilla koneilla listataan projektit sekä niiden sisällä koneen miehistön lukumäärä sekä mukana olleet vaunut ja lisälaitteet. Yhteenvetonäkymässä työvuoron tietoja voidaan vielä muokata tarvittaessa hyvin laajastikin ennen sen lähettämistä palvelimelle. Esimerkiksi työvuoron, projektien ja ajokoodien alkamis- ja päättymisaikoja voidaan muuttaa tai tarvittaessa myös lisätä työvuorolle uusia

projekteja tai projekteille ajokoodeja. Kun työvuoro on tässä näkymässä tarkistettu, tarvittavat muokkaukset tehty ja työvuoron, projektien ja ajokoodien kellonajat ovat laillisia eli esimerkiksi niiden välillä ei ole aukkoja, se voidaan lähettää palvelimelle. Jos palvelimeen ei saada yhteyttä, työvuoro säilytetään lähettämättömien työvuorojen listassa selaimen välimuistissa, mistä se voidaan myöhemmin lähettää.

3. JÄRJESTELMÄN KEHITYSTAVOITTEET

Tämän työn pääasiallisena kehitystavoitteena on tuoda järjestelmän kirjausprosessin informaationkulkuun reaaliaikaisuutta siten, että esimiehet voivat seurata raportointisovelluksessa reaaliajassa suoritekirjauksia, joita konemiehet tekevät kirjaussovelluksessa. Ilman tätä reaaliaikaisuutta informaatiota työvuoroista saadaan järjestelmän muille osille vasta, kun konemies on suorittanut työvuoron loppuun ja lähettänyt tiedot sen vaiheista palvelimelle. Työvuorojen informaationkulkua halutaan parantaa, jotta saataisiin selkeämpi ja ajantasaisempi käsitys esimerkiksi suunniteltujen työvuorojen toteutumisesta, työvuorojen yleisestä etenemisestä tai niiden mahdollisista ongelmatilanteista.

Reaaliaikaista informaationkulkua hyödyntävänä keskeisenä ominaisuutena järjestelmään halutaan raportointisovelluksen puolelle työvuorojen seurantasivu. Tämä sivu halutaan jakaa kahteen eri näkymään: kaavionäkymään ja listausnäkymään.

Kaavionäkymän tavoitteena on työvuorojen seuranta sovelletun gantt-kaavion muodossa, jossa vaaka-akselilla on aikajana ja koneet ovat riveittäin. Tästä kaaviosta olisi nähtävillä konekohtaisesti aikajanalla työvuorojen, projektien ja ajokoodien alku- ja loppuajat. Kaavion halutaan päivittyvän reaaliajassa ja kullakin hetkellä aktiivisesti kirjausta suorittavat eli online-tilassa olevat koneet halutaan näkyvän korostetusti.

Listausnäkymän tavoitteena on yksinkertaisesti nähdä listaus koneiden tilojen muuttumisesta ja niiden ajankohdista sekä listaus online-tilassa olevista koneista. Eli listaukseen on tarkoitus ilmestyä reaaliaikaisesti uusi rivi aina, kun jokin kone aloittaa tai lopettaa työvuoron, projektin tai ajokoodin. Sekä listausnäkymässä että kaavionäkymässä yhtä aikaa näkyvät koneet halutaan pystyä tarvittaessa valitsemaan suodatusvalikon avulla.

Keskeisenä haasteena reaaliaikaisen viestinnän toteutumiselle on työkoneiden mobiiliverkkoyhteys. Ratatyötä tehdään ajoittain alueilla, joissa verkkoyhteyden laatu on huono ja työkoneen verkkoyhteys tulee olemaan ajoittain poikki. Tämä seikka on siis otettava huomioon ja toteutus suunniteltava siten, että yhteyksien hallinta palvelimella toimii oikein yksittäisistä yhteyksien katkeamisista huolimatta ja että seurantakäyttöliittymä on riittävän informatiivinen työkoneiden verkkoyhteyksien tilan suhteen.

Kaiken kaikkiaan työllä on teknisiä tavoitteita kahdelta erilaiselta osa-alueelta. Reaaliaikaiselle kaksisuuntaiselle tiedonkululle halutaan löytää ja ottaa käyttöön ratkaisu, joka on verkkoyhteyden ja laitteiston kuormituksen suhteen tehokas, joka on järkevästi toteutettavissa olemassa olevaan järjestelmään ja joka mahdollistaa jatkokehitettävyysmah-

dollisuuksia siten, että ratkaisua voitaisiin hyödyntää myös tulevaisuudessa uusien ominaisuuksien kehittämisen yhteydessä. Seurantakäyttöliittymän suunnittelussa ja toteutuksessa halutaan panostaa hyvään käytettävyyteen ja informatiivisuuteen.

4. JÄRJESTELMÄN TEKNOLOGIAT

Suoritekirjausjärjestelmä on rakennettu useita teknologioita hyödyntäen. Tässä luvussa esitellään tämän työn kannalta merkittävimpien web-teknologioiden taustaa ja tehtäviä järjestelmässä.

4.1 HTTP

HTTP (Hypertext Transfer Protocol) on World Wide Webiä (WWW) varten kehitetty tiedonsiirtoprotokolla. Se on tilaton protokolla, joka toimii verkkoprotokollien OSI-mallin sovelluskerroksella. [3] OSI-malli jakaa verkkoprotokollat seitsemään eri tasoon siten, että kukin protokolla käyttää jonkin omaa tasoaan alemman tason protokollan palveluita toimiakseen [4]. HTTP käyttää kuljetuskerroksella toimivan TCP-protokollan palveluita, eli HTTP-tiedonsiirto tapahtuu TCP-yhteyden välityksellä. TCP (Transmission Control Protocol) on luotettava datapakettien välitysprotokolla verkkoisäntien välillä [5].

HTTP-protokollan toiminta perustuu viestien vaihtoon asiakasohjelman ja palvelimen välillä pyyntöjen ja vastausten muodossa. Asiakasohjelma lähettää TCP-yhteyden välityksellä palvelimelle HTTP-pyynnön, johon palvelin vastaa lähettämällä asiakasohjelmalle pyyntöä vastaavan resurssin. [6] HTTP-protokollaa käytettäessä palvelin ei siis voi omaaloitteisesti lähettää dataa asiakasohjelmalle, vaan asiakasohjelman on ensin tehtävä pyyntö.

HTTP-pyyntö alkaa request-line -rivillä, johon kuuluu käytettävä pyynnön tarkoitusta indikoiva HTTP-metodi, kohderesurssin indikoiva URI-osoite ja HTTP-protokollan versio. Request-line -riviä seuraa joukko otsikkorivejä, jotka sisältävät lisätietoja, kuten tietoa asiakasohjelmasta tai pyynnön sisältämän datan esitystyylistä. Otsikkorivien jälkeen pyynnössä on tyhjä rivi, jota seuraa viestin runko eli sen sisältämä data. [3] Esimerkki HTTP-pyynnöstä on esitetty ohjelmassa 1.

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Ohjelma 1. *Esimerkki HTTP-pyynnöstä [3]*

Palvelimen vastaus asiakasohjelman pyyntöön alkaa status-line -rivillä, johon kuuluu HTTP-protokollan versio, pyynnön onnistumisesta tai epäonnistumisesta kertova status-

koodi ja tekstimuotoinen kuvaus statuskoodista. Myös status-line -riviä seuraa valinnainen joukko otsikkorivejä, jotka antavat ylimääräistä tietoa vastauksesta. Otsikkorivejä seuraa tyhjä rivi, jonka jälkeen vastaus sisältää mahdollisen mukana välittyvän datan. [3] Ohjelmassa 2 on esitetty esimerkki HTTP-vastauksesta.

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain

Hello World! My payload includes a trailing CRLF.
```

Ohjelma 2. *Esimerkki HTTP-vastauksesta [3]*

Suoritekirjausjärjestelmä on web-pohjainen järjestelmä, joten HTTP toimii siinä tiedonsiirtoprotokollana. Sen avulla tietoa saadaan välitettyä palvelimen ja kirjaus- sekä raportointisovelluksien välillä.

4.2 Play Framework

Suoritekirjausjärjestelmän backend-palvelin on toteutettu Scala -ohjelmointikielellä ja Play Framework -sovelluskehyksellä. Play Framework on web-sovelluskehys, jota voidaan käyttää Java- ja Scala -ohjelmointikielillä. Se tarjoaa web-sovellusten kehittämiseen tarvittavat komponentit, kuten HTTP-palvelimen, lomakkeiden hallinnan ja reititysmekanismin. Sen arkkitehtuuri on suunniteltu kevyeksi, tilattomaksi ja web-yhteensopivaksi. [7]

Play Framework on toteutettu MVC-arkkitehtuurin (Model-View-Controller) mukaisesti, mikä tekee siitä toimivan ratkaisun web-ympäristöön. MVC-malli jakaa järjestelmän vastuut kolmeen osa-alueeseen ja jokaisen osa-alueen toteutuksesta vastaa erillinen komponentti. Näitä osa-alueita ovat malli, joka vastaa järjestelmässä tietomalleihin eli dataan ja se käsittelyyn liittyvistä asioista, näkymä, joka vastaa käyttöliittymän hallinnasta, sekä kontrolleri, joka vastaa tapahtumien, kuten käyttäjän suorittamien toimintojen ja niiden vaikutusten hallinnasta järjestelmässä. [8]

Play Frameworkista tekee tehokkaan ja skaalautuvan sen monisäikeinen toteutus, joka on luotu Akka- ja Akka Steams -teknologioita hyödyntäen. Akka on Scala- ja Java -ohjelmointikielille kehitetty joukko kirjastoja, jotka on tarkoitettu monisäikeisten, skaalautu-

vien ja virhesietoisten järjestelmien kehittämiseen. Akka tarjoaa helppokäyttöisen ratkaisun rinnakkaisuuden toteuttamiseen siten, ettei kehittäjän tarvitse huolehtia alhaisen tason yksityiskohdista. Se hyödyntää niin kutsuttua Actor-mallia, joka tarjoaa abstraktiokerroksen rinnakkaisten komponenttien kommunikaatioon ja skaalaamiseen. [9]

Akka Actor-malli on suunniteltu helpoksi tavaksi toteuttaa rinnakkaisia komponentteja keskittyen niiden yksilöllisiin tehtäviin ja niiden väliseen kommunikaatioon rinnakkaisuuden tarkkojen teknisten toteutusmekanismien sijaan. Mallin toiminta perustuu Actor-olioihin, joilla on tila ja jotka reagoivat vastaanottamiinsa signaaleihin ja lähettävät signaaleita muille Actor-olioille. Signaalipohjaisesta toimintatavasta seuraa se, että jokaista Actor-oliota voidaan suorittaa samanaikaisesti, eikä järjestelmän toisen osan kanssa kommunikointi keskeytä tai viivytä suoritusta kutsuvassa järjestelmän osassa. [9]

4.3 Angular

Suoritekirjausjärjestelmän frontend-komponentit eli kirjaus- ja raportointisovellukset on toteutettu Angularilla. Angular on Googlen kehittämä TypeScript-ohjelmointikieleen pohjautuva frontend web-sovelluskehys. Se noudattaa MVC-mallia, mikä tekee sitä käyttävistä sovelluksista helposti laajennettavia ja ylläpidettäviä. [10]

Angular-sovelluksia kehitetään TypeScript-ohjelmointikielellä. TypeScript on Microsoftin kehittämä JavaScriptin ominaisuuksia laajentava ohjelmointikieli. Se on suunniteltu ratkaisemaan ongelmia, jotka liittyvät JavaScriptin käyttämiseen suuren kokoluokan järjestelmissä. Tärkeimpänä ominaisuutenaan se tuo JavaScriptiin tyypitetyt muuttujat. TypeScript-ohjelmat käännetään puhtaaksi JavaScriptiksi, jolloin niitä voidaan suorittaa monilla eri alustoilla, kuten verkkoselaimissa. [11]

Angular-sovelluksen kehityksessä keskeisiä elementtejä ovat direktiivit, komponentit ja palvelut (service). Direktiivit ovat yksi tapa liittää Angular-sovelluksen tietomalli ja näkymä toisiinsa. Niitä käytetään esimerkiksi sovelluksen tietomallin sisällön esittämiseen sovelluksen näkymissä dynaamisesti. Yleisimmin käytetyt Angularin valmiit direktiivit ovat ehtodirektiivi *ngIf,* jolla sisältö esitetään, jos sille asetettu ehto täyttyy sekä toistodirektiivi *ngFor*, jolla voidaan esittää esimerkiksi taulukko objekteja samalla toisteisella rakenteella. [10]

Angular-sovelluksessa komponentti on looginen kokonaisuus, joka käsittää järjestelmän yksittäisen osan tietomallin, sovelluslogiikan ja näkymän. Komponenttien avulla suurenkin järjestelmän kokonaisuuden hallinta on helppoa, sillä ne ovat toisistaan erillisiä kokonaisuuksia, eivätkä vaikuta toistensa toimintaan. Komponentteja voidaan käyttää järjestelmän eri osissa joustavasti ja erilaisina yhdistelminä. [10]

Palvelut ovat Angularin tapa tarjota järjestelmän rakennuspalikoille, kuten komponenteille, yhteistä toiminnallisuutta ja yhteisiä tietomalleja. Niitä käytetään niin kutsutun riippuvuusinjektion (dependency injection) avulla. Palvelu voidaan injektoida esimerkiksi sitä tarvitsevaan komponenttiin, jolloin komponentissa voidaan hyödyntää sen tarjoamaa toiminnallisuutta ja päästä käsiksi sen sisältämään tietoon esimerkiksi jonkin järjestelmän osan tilasta. Yksittäinen palvelu on kaikille sitä käyttäville komponenteille sama, joten niillä kaikilla on pääsy esimerkiksi samaan muuttuvaan tilatietoon. [10]

4.4 JSON

JSON (JavaScript Object Notation) on yksinkertainen ja tehokas dataobjektien rakenteen määrittelykieli. Sen yksinkertaisuus tekee siitä helposti luettavan sekä ihmiselle että tietokoneelle, minkä vuoksi se on suosittu tapa välittää dataa eri järjestelmien tai saman järjestelmän osien välillä etenkin web-ympäristössä. [12] Suoritekirjausjärjestelmässä JSON:ia käytetään HTTP-pyyntöjen ja vastausten yhteydessä välittämään dataa palvelimen ja frontend-sovellusten välillä.

JSON-määrittely perustuu kahteen erilaiseen datarakenteeseen, dataobjekteihin ja datataulukoihin, mikä tekee siitä helposti luettavan ja tiedon välittämiseen vaadittavan datamäärän suhteen tehokkaan. Erilaisten rakennetyyppien vähäisestä määrästä huolimatta se on tehokas tapa dataobjektien monimutkaiseenkin määrittelyyn. [12]

Dataobjektit ovat avain-arvopareja, joita kuvataan niitä ympäröivillä aaltosulkeilla. Dataobjekteja voidaan määritellä myös sisäkkäin, eli dataobjektin yksittäisen avaimen arvona voi olla toinen dataobjekti. [12]

Datataulukot ovat järjestettyjä listoja, joiden arvoja voidaan tarkastella niiden järjestyslukuihin perustuen. Datataulukot kuvataan hakasulkeilla, joiden sisään taulukon arvot on sijoitettu pilkulla erotettuina. [12] Ohjelmassa 3 on esitetty esimerkki JSON-muotoon määritellystä objektista. Kuvattu objekti on määritelty aaltosulkeiden sisään ja se sisältää avain-arvopareja, joista viimeisen arvona on taulukko.

```
{
   ”id”: 1,
   ”otsikko”: ”esimerkki”,
   ”mittaustulokset”: [
         3,
         4,
         5
   ]
}
```

Ohjelma 3. *Esimerkki JSON-muodossa määritellystä objektista*

JSON-muotoiset dokumentit esitetään usein usealla rivillä sisennyksiä käyttäen, mikä tekee niistä helposti luettavia ihmiselle. JSON on yhteensopiva erityisesti JavaScript-ohjelmointikielen kanssa, sillä sen rakenne muistuttaa JavaScript-objektin rakennetta. Kuitenkin sen yksinkertaisen rakenteen vuoksi se on helposti tulkittavissa dataobjekteiksi myös kaikilla muilla ohjelmointikielillä. [12]

5. REAALIAIKAISEN SEURANNAN SUUNNITTELU JA TOTEUTUS

Reaaliaikaisen seurannan toteuttamista varten vertailtiin erilaisia menetelmiä teorian tasolla. Näistä valittiin toteutettavaksi kehitystavoitteet parhaiten täyttävä vaihtoehto. Tässä luvussa toteutetaan menetelmien vertailu, käydään läpi protokollan toteutusprosessi valitulla menetelmällä ja esitellään reaaliaikaiselle seurannalle toteutettu käyttöliittymä.

5.1 Reaaliaikainen kaksisuuntainen kommunikaatio

Normaalisti tiedonsiirto asiakas-palvelinpohjaisissa järjestelmissä tapahtuu HTTP-protokollan mukaisesti TCP-yhteyden välityksellä. Kirjausjärjestelmän tapauksessa HTTP-protokollan mukaisena asiakasohjelmana toimii verkkoselain. Palvelinosapuolena toimii kirjausjärjestelmän backend-palvelin.

Työvuorojen tilojen halutaan päivittyvän raportointisovelluksessa automaattisesti, mikä vaatii toiminnollisuuden, jonka avulla palvelin voi omasta aloitteestaan lähettää asiakasohjelmalle tietoa muutoksista niiden tapahtuessa. HTTP-protokollan normaalin käyttötavan avulla tämä ei ole mahdollista, joten sen perustoimintamalli ei ole riittävä kirjausjärjestelmän raportointisovelluksen kehitystavoitteiden kannalta.

Kirjaussovelluksen osalta ei ole tässä työssä kehitettävien ominaisuuksien puitteissa tarvetta sille, että palvelin voisi lähettää asiakasohjelmalle tietoa, sillä kirjaussovelluksen on tarkoitus lähettää vain päivityksiä työvuoron tapahtumista palvelimelle. Näin ollen HTTP-protokollan käyttäminen on riittävä ratkaisu kirjaussovelluksen tarpeisiin.

Tarvittavaan kaksisuuntaiseen viestien välitykseen web-ympäristössä on olemassa erilaisia ratkaisuja. Näitä käydään läpi kappaleissa 5.1.1, 5.1.2 ja 5.1.3. Kappaleessa 5.1.4 näitä ratkaisuja vertaillaan ja niiden välillä tehdään valinta kirjausjärjestelmässä käytettäväksi ratkaisuksi.

5.1.1 HTTP-protokollan soveltaminen

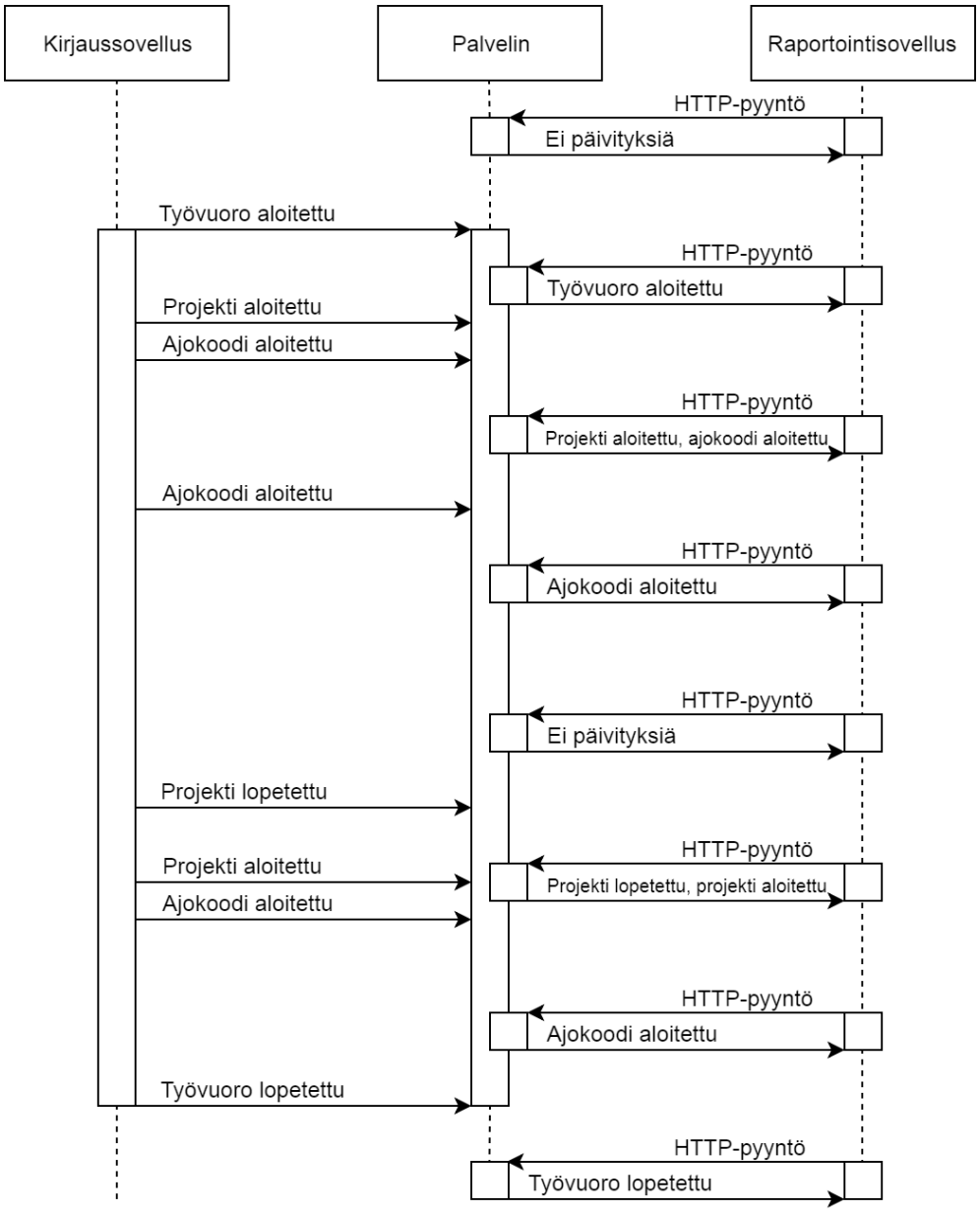
HTTP-protokollan mukaisesti palvelin ei voi toimia aloittavana osapuolena yhteyden muodostuksessa asiakasohjelman kanssa. HTTP-pyyntöjen toimintatavasta johtuen HTTP-protokollaa on kuitenkin mahdollista soveltaa reaaliaikaisten tai lähes reaaliaikaisten päivitysten saamiseksi palvelimelta. [13]

HTTP-pyyntö-vastaus –prosessi toimii siten, että asiakasohjelma lähettää palvelimelle TCP-yhteyden välityksellä HTTP-pyynnön ja jää odottamaan vastausta. Palvelin prosessoi asiakasohjelman lähettämän pyynnön ja lähettää takaisin vastauksen. Palvelimen prosessoidessa pyyntöä yhteys säilyy avoimena. Palvelimen lähetettyä vastauksen yhteys sulkeutuu ja palvelin ei voi enää lähettää asiakasohjelmalle uutta dataa ennen kuin asiakasohjelma tekee uuden HTTP-pyynnön. [6]

Toteutuksen kannalta yksinkertaisin HTTP:hen pohjautuva ratkaisu on se, että asiakasohjelma lähettää palvelimelle määritellyn ajan välein normaalin HTTP-pyynnön, jolla se pyytää uusia päivityksiä. Tämän määritellyn aikavälin pituus riippuu siitä, millaisella viiveellä päivityksiä korkeintaan halutaan vastaanottaa. Tällainen HTTP polling-menettely ei mahdollistaisi täysin reaaliaikaista päivitysten välittymistä, mutta pyyntöjen aikavälin ollessa esimerkiksi kymmenen sekuntia, maksimiviive ei todennäköisesti olisi kirjausjärjestelmän kannalta liian suuri.

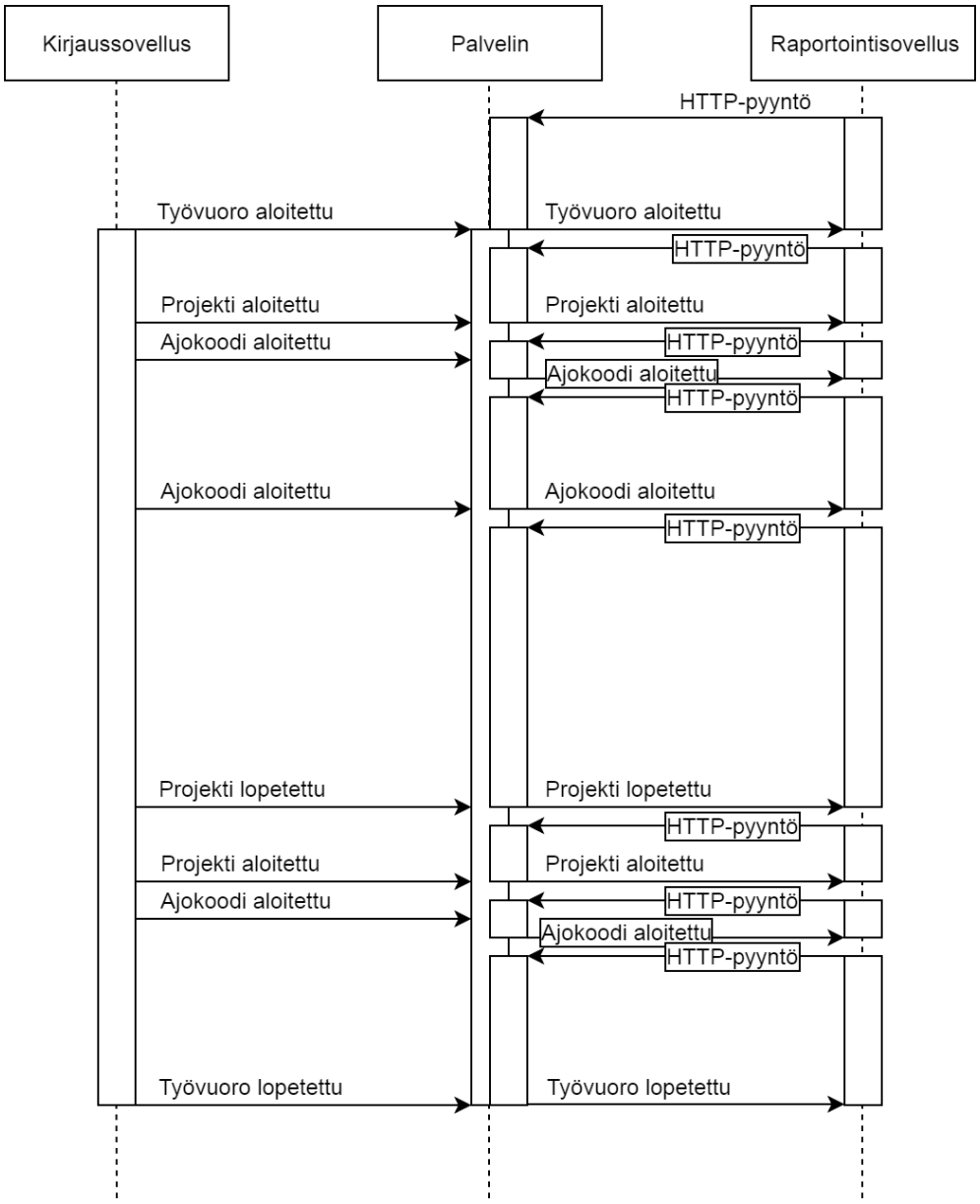
Sen lisäksi, että tämä ratkaisu ei mahdollistaisi täysin reaaliaikaista päivitysten välittymistä, sen huonona puolena on resurssien suuri kulutus siirrettävään datamäärään nähden. Jos aktiivisia asiakasohjelmia on samaan aikaan useita, jatkuva pyyntöjen vastaanottaminen varaa paljon palvelimen internetkaistaa ja prosessointitehoa [13]. Lisäksi pyynnöt saavat usein tyhjän vastauksen ja ovat näin ollen teoriassa turhia, jos uusia päivityksiä ei pyynnön hetkellä olekaan saatavilla.

Sitä, kuinka kommunikaatio kirjausjärjestelmän osien välillä tapahtuisi tätä menetelmää soveltaen käytännössä, on kuvattu kuvan 4 sekvenssikaaviossa. Kaaviosta käy ilmi, että raportointisovellus saisi tiedon päivityksistä hyvin vaihtelevalla viiveellä ja joissain tapauksissa se voisi vastaanottaa useammankin päivityksen yhdellä pyynnöllä.



Kuva 4. *Kaksisuuntaisen kommunikaation HTTP polling-toteutuksen sekvenssikaavio*

HTTP-protokollaa soveltava menetelmä reaaliaikaisen datan lähettämiseen palvelimelta asiakasohjelmalle on niin kutsuttu ”HTTP Long Polling” eli HTTP-vastauksen viivyttäminen. Tässä menetelmässä palvelin yrittää viivyttää asiakasohjelman pyyntöön vastaamista ja pitää yhteyttä avoimena niin kauan, että sillä on jotain uutta dataa, jota tarjota asiakasohjelmalle tai pyyntö on ollut avoimena tietyn maksimiaikarajan sen sijaan, että se vastaisi pyyntöön heti. Näin palvelimella on aina odottava pyyntö, johon se voi vastata ja uusi data välittyy asiakasohjelmalle viiveettä. Saatuaan vastauksen asiakasohjelma lähettää heti uuden Long Polling –pyynnön, jolloin palvelimella on taas uusi pyyntö odottamassa. [13] Järjestelmän osien välinen kommunikaatio tätä menetelmää soveltaen on esitetty kuvassa 5.



Kuva 5. *Kaksisuuntaisen kommunikaation HTTP Long Polling –toteutuksen sekvenssikaavio*

HTTP Long Polling –menetelmä mahdollistaa siis näennäisesti sen, että palvelin voi oma-aloitteisesti lähettää dataa asiakasohjelmalle. Tällä menetelmällä ei varsinaisesti ole yhtä merkittäviä huonoja puolia, kuin HTTP-pyyntöjen lähettämisellä tasaisin aikavälein, mutta jos esimerkiksi päivityksiä tapahtuu riittävän harvoin, asiakasohjelma voi aikakatkaista pyyntöjä, jolloin pitkällä aikavälillä voi tapahtua paljon tarpeettomiakin pyyntöjä. [13] Lisäksi voidaan ajatella, että menetelmä ei varsinaisesti noudata oikein HTTP-protokollan käytäntöjä.

HTTP-protokollaan pohjautuvissa menetelmissä on myös joitain yhteisiä huonoja puolia. Tällaisia ovat esimerkiksi se, että palvelimen täytyy pitää avoinna useampaa TCP-yhteyttä jokaista asiakassovellusta kohden ja se, että jokainen viesti sisältää HTTP-protokollan vaatiman otsikkodatan, mikä omalta osaltaan kuormittaa yhteyttä [14].

5.1.2 WebSocket-protokolla

WebSocket-protokolla mahdollistaa kaksisuuntaisen kommunikoinnin palvelimen ja asiakasohjelman välillä hyvin pienellä viiveellä. Se on suunniteltu korvaamaan olemassa olevat HTTP:hen pohjautuvat kaksisuuntaisen kommunikoinnin menetelmät ja se tarjoaa niille yksinkertaisemman ja kevyemmän vaihtoehdon, sillä sen avulla liikenne kulkee molempiin suuntiin vain yhden TCP-yhteyden välityksellä. WebSocket-protokollaa voidaan sen monipuolisuuden ansiosta käyttää hyvin monenlaisissa reaaliaikaisissa sovelluksissa, kuten peleissä tai keskustelualustoissa. [14]

WebSocket-protokollan toiminta koostuu kättelyvaiheesta ja datansiirtovaiheesta. Kättelyviestit on suunniteltu siten, että ne ovat luettavissa HTTP-pohjaisille palvelinohjelmille, eli ne ovat käytännössä HTTP-protokollan mukaisia viestejä. Siten asiakasohjelma voi käyttää samaa palvelimen porttia sekä normaaliin HTTP-liikenteeseen että WebSocket-liikenteeseen. WebSocket-asiakkaan kättelyviesti on HTTP Upgrade -pyyntö, joka sisältää joukon WebSocket-yhteyden muodostamiselle välttämättömiä ja valinnaisia otsikkorivejä. Asiakkaan kättelyviesti noudattaa ohjelman 4 mukaista kaavaa. [14]

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Ohjelma 4. *Esimerkki WebSocket-asiakkaan kättelyviestistä [14]*

Palvelimen kättelyviesti on HTTP-vastaus ja se sisältää myös WebSocket-yhteyden avaamiseen liittyviä otsikkorivejä. Palvelimen kättelyviesti on ohjelman 5 mukainen. [14]

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

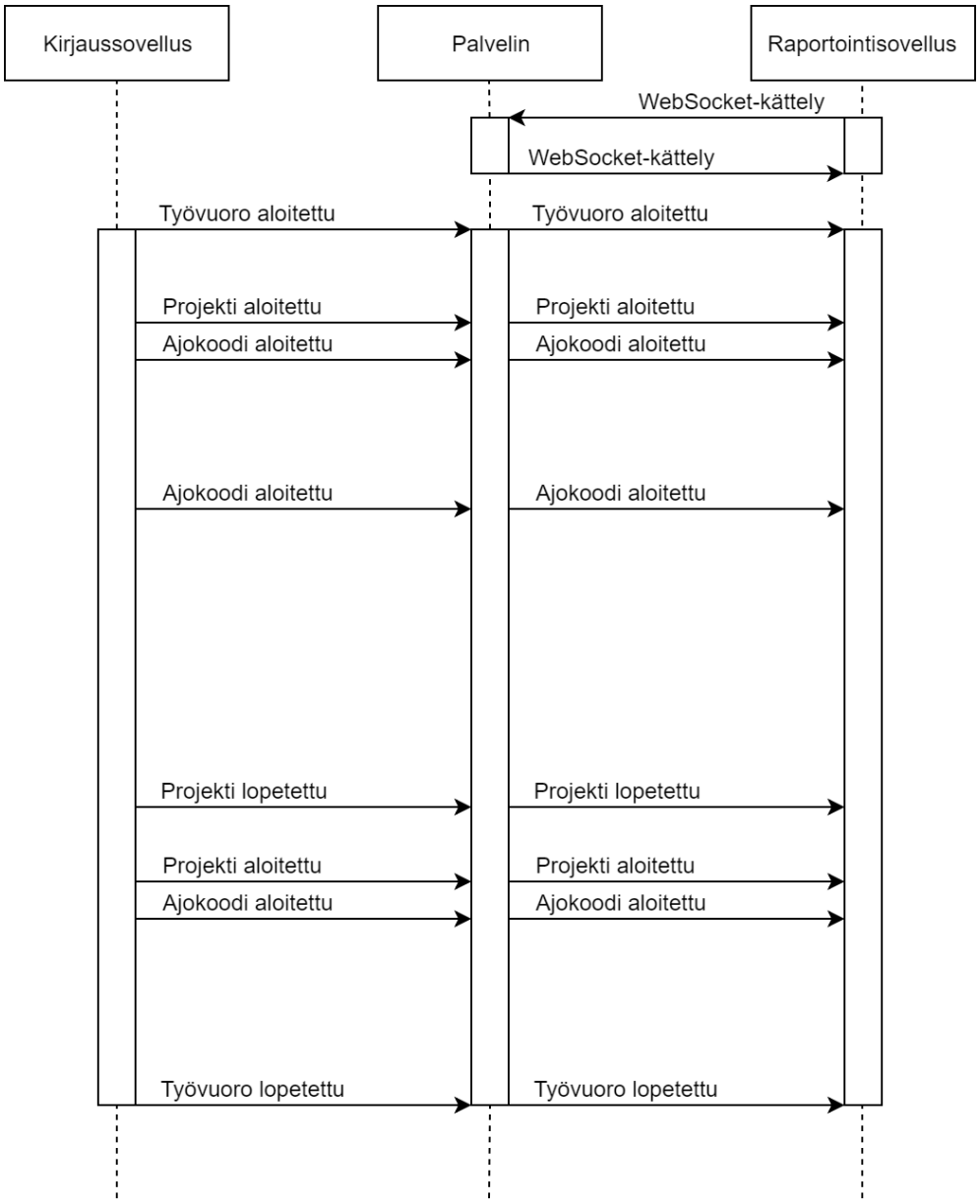
Ohjelma 5. *Esimerkki WebSocket-palvelimen kättelyviestistä [14]*

WebSocket-yhteys muodostetaan ja datansiirtovaihe aloitetaan, jos asiakasohjelma toteaa, että palvelimen kättelyviestin ensimmäisen rivin HTTP-statuskoodi on *101 Switching Protocols* ja otsikkorivin *Sec-WebSocket-Accept* arvo on odotettu. Palvelin laskee otsikon *Sec-WebSocket-Accept* arvon tietyn algoritmin mukaisesti asiakasohjelman lähettämän otsikkorivin *Sec-WebSocket-Key* arvosta. [14]

Datansiirtovaiheessa sekä palvelin että asiakasohjelma voivat vapaasti lähettää toisilleen dataa. WebSocket-yhteyden välityksellä lähetettäviä protokollan mukaisia datakokonaisuuksia kutsutaan viesteiksi. Yhteyden matalammalla tasolla viestit koostuvat yhdestä tai useammasta kehyksestä. Kehyksellä on tietty tyyppi ja tämä tyyppi on yhteinen kaikille saman viestin kehyksille. Tällaisia tyyppejä voivat olla esimerkiksi tekstidata UTF-8-enkoodatun tekstin välitykseen, binääridata, jonka tulkinta on ohjelmille vapaasti toteutettavissa ja hallintakehykset, jotka on tarkoitettu protokollatason signaalien välitykseen. [14]

WebSocket-yhteys suljetaan sulkemiskättelyllä. Tämän kättelyn voi aloittaa kumpikin osapuoli. Sulkemiskättelyn aloittava osapuoli lähettää hallintakehyksen, joka sisältää sulkemissignaalin. Tämän kehyksen vastaanotettuaan toinen osapuoli lähettää takaisin oman sulkemiskehyksensä. Kun osapuoli on lähettänyt sulkemiskehyksen, se ei lähetä enää uutta dataa ja kun osapuoli on vastaanottanut sulkemiskehyksen, osapuoli ei enää huomioi sille mahdollisesti lähetettyjä uusia viestejä. Kun ensimmäinen sulkemiskehyksen lähettänyt osapuoli vastaanottaa toisen osapuolen sulkemiskehyksen, se voi turvallisesti sulkea yhteyden. Tämä yhteyden sulkemisjärjestely on turvallinen myös siinä tapauksessa, että molemmat osapuolet lähettävät sulkemiskehyksensä samaan aikaan. [14]

Sitä, kuinka data välittyisi kirjausjärjestelmän osien välillä WebSocket-protokollaa hyödyntäen, on kuvattu kuvan 6 sekvenssikaaviossa. Kaaviosta voidaan todeta, että raportointisovellus saisi tämän protokollan avulla tiedon kirjaussovelluksista lähetettävistä päivityksistä käytännössä reaaliajassa.



Kuva 6. *Kaksisuuntaisen kommunikaation WebSocket-toteutuksen sekvenssikaavio*

WebSocket-protokolla on suunniteltu sillä periaatteella, että otsikkodataa tulisi olla mahdollisimman vähän. Käytännössä se on TCP-protokollan päällä toimiva taso, joka tuo TCP-yhteyden välityksellä kulkevalle datalle joitain tarpeellisia HTTP-liikenteelle tyypillisiä ominaisuuksia, kuten alkuperään pohjautuvan tietoturvamekanismin. Tällä tavoin WebSocket-yhteyden välityksellä kulkeva data on web-liikenteen vaatimukset huomioiden mahdollisimman lähellä puhtaasti TCP-yhteyden kautta kulkevaa dataa. WebSocket-protokolla tarjoaa siis yksinkertaisen tavan siirtää dataa HTTP:n rinnalla minimaalisella ylimääräisellä otsikoinnilla. [14]

WebSocket-protokollan käyttö verkkoselaimissa onnistuu JavaScriptin WebSocket-rajapinnan avulla. Tätä varten JavaScriptissä on olemassa helppokäyttöinen WebSocket-

luokka. WebSocket-luokan avulla voidaan esimerkiksi lähettää WebSocket-viestejä, sulkea yhteys tai määrittää, mitä asiakasohjelma tekee esimerkiksi vastaanottaessaan WebSocket-viestin tai kun WebSocket-yhteys avataan. [15]

Play Frameworkista löytyy valmis tuki WebSocket-protokollan palvelinpuolen toteuttamiseen. WebSocket-yhteyksien hallinta onnistuu sen Akka Actor-mallin avulla. Actorien avulla onnistuu esimerkiksi yhteyksien avaaminen ja eri viestityyppien vastaanottaminen, yhteyden sulkeminen, viestien lähettäminen ja yhteyden sulkeutumisen havaitseminen. [16]

WebSocket-protokolla soveltuisi kirjausjärjestelmän reaaliaikaisen kaksisuuntaisen viestinnän tarpeisiin hyvin. Raportointisovelluksen kannalta sen tuoma helppo viestin lähettäminen palvelimelta asiakasohjelmalle on hyödyllinen ominaisuus. Lisäksi protokollan keveys otsikkodatan määrän suhteen on eduksi, jos palvelimeen on yhteydessä suuri määrä asiakasohjelmia. Ratkaisulle on tarjolla pohjatoteutuksia sekä palvelimen, että asiakassovelluksen puolelle, joten se olisi toteutettavuutensakin kannalta hyvä. Play Frameworkin palvelintoteutuksessa mahdollistamaa yhteyden sulkeutumisen havaitsemista voitaisiin hyödyntää myös kirjaussovelluksen kannalta siten, että koneiden online-tilaa voitaisiin helposti seurata.

5.1.3 Server Sent Events (SSE)

Server Sent Events (SSE) on HTML5:n rajapinta, joka on suunniteltu helppokäyttöiseksi tavaksi mahdollistaa palvelimen verkkoselaimelle lähettämät tapahtumat. Tätä rajapintaa tukevat useimmat verkkoselaimet Internet Explorer- ja Edge -selaimia lukuun ottamatta. SSE on suunniteltu nimenomaan datan lähettämiseen palvelimelta asiakasohjelmalle, joten se ei itsessään tarjoa tapaa datan lähettämiseen asiakasohjelmalta palvelimelle. SSE-rajapinnan avulla saadaan siis avattua yksisuuntainen viestintäkanava palvelimelta asiakasohjelmalle. [17]

Alhaisemmalla tasolla SSE:n toteutus toimii siten, että asiakasohjelma lähettää yhden HTTP-pyynnön, jonka kautta palvelin lähettää kaikki päivitykset HTTP-vastauksen osina. Asiakasohjelman kannalta kyseessä on käytännössä helppokäyttöinen JavaScript-rajapinta, jossa selainohjelman sisäinen toteutus hoitaa yhteyden tekniset yksityiskohdat ja kehittäjä voi keskittyä päivitysviestien käsittelyyn. [18]

SSE-rajapinnan asiakaspuolen toteutus löytyy valmiina useimmista nykyaikaisista verkkoselaimista, mutta sen käyttö vaatii myös palvelinohjelmiston, joka kykenee käsittelemään asiakasohjelmilta tulevat kanavan avauspyynnöt ja julkaisemaan oikeanmuotoisia

tapahtumapäivityksiä. Tämä onnistuu käytännössä millä vain palvelinpuolen ohjelmointikielellä, sillä SSE-tekniikka pohjautuu määrätyn tyyppisiin HTTP-pyyntöihin ja -vastauksiin. [17]

SSE-tekniikassa palvelinpuolen toteutuksen tulee vastata asiakasohjelman pyyntöihin vastauksella, jonka Content-Type -otsikkorivin arvo on *text/event-stream*. Tällainen vastaus koostuu avain-arvopareista ja päättyy kahteen rivinvaihtoon. Hyväksyttyjä avaimia ovat:

- *data*: asiakasohjelmalle lähetettävä tekstimuotoinen viesti,
- *event*: vapaavalintainen tapahtumatyyppi, normaaleja tapahtumatyyppejä ovat *open*, *message* ja *error*,
- *id*: vapaavalintainen päivityksen tunniste, jota voidaan käyttää yksilöimään päivitykset,
- *retry*: aika millisekunneissa, jonka jälkeen asiakasohjelma saa tiedustella seuraavaa päivitystä.

Asiakasohjelman puolella verkkoselain hoitaa vastausten tulkinnan ja uusien kyselyiden tekemisen. Kehittäjä voi lukea päivitykset SSE-rajapinnan tarjoaman *EventSource*-olion avulla. [17]

Play Frameworkilla onnistuu myös SSE:n palvelinpuolen toteuttaminen. WebSocket-toteutuksen tapaan tämäkin tapahtuu Akka Actor-mallin avulla. Jokaiselle yhteydelle voidaan määrittää oma Actor-objekti ja viestien lähettäminen asiakasohjelmille tapahtuu välittämällä viesti yhteyden Actor-objektille. [18]

Kirjausjärjestelmän kehitystavoitteiden kannalta Server Sent Events olisi toimiva ratkaisu. Raportointisovellukselle riittää, että viestintäkanava voidaan avata ja palvelimelta voidaan vastaanottaa päivitystapahtumia. Toteutettavuutensa puolesta SSE-ratkaisu on hyvä, sillä valmiita rajapintoja ja pohjatoteutuksia löytyy sekä asiakas-, että palvelinpäästä. EventSource-rajapinnan puuttuminen Internet Explorer- ja Edge-selaimista rajoittaisi kuitenkin toteutustavan käyttökelpoisuutta jonkin verran. Kirjaussovelluksen puolella kirjaustapahtumien lähettäminen palvelimelle voitaisiin toteuttaa perinteisesti HTTP-protokollan avulla.

5.1.4 Ratkaisujen vertailu

Kirjausjärjestelmän pääasiallinen kehitystavoite ja toiminnallinen vaatimus reaaliaikaisen päivitysten lähettämisen suhteen on se, että palvelimelta saadaan päivitysviestien muodossa uuttaa dataa raportointisovellukselle sen tullessa kirjaussovellukselta palvelimen

saataville. Kaikki edellä kuvatut ratkaisut tekevät tämän mahdolliseksi vähintään pienellä viiveellä, mutta ratkaisujen välillä on eroja esimerkiksi niiden datan- ja resurssienkäytön tehokkuuden suhteen.

Päivitysten välittämisen lisäksi palvelimen tulee tietää ja välittää raportointisovellukselle tieto siitä, mitkä koneet ovat parhaillaan kirjaamassa suoritetta eli online-tilassa. Tämäkin onnistuu kaikilla edellä kuvatuista ratkaisuista, mutta eri menetelmien tapaa hallita avoimia yhteyksiä voidaan mahdollisesti hyödyntää toteutuksessa.

HTTP-protokollan soveltaminen on ratkaisuista huonoin vaihtoehto. Ainoana hyvänä puolena voidaan toteutettavuuden kannalta nähdä se, että ratkaisu ei vaadi uusien teknologioiden käyttöönottoa ja opettelemista. Sen avulla tarvittavat toiminnot toteutettaisiin samoilla web-kehityksen perustekniikoilla, kuin muukin tiedonsiirto järjestelmässä. Päivitykset saataisiin raportointisovellukselle polling-pyynnöillä, joita tehtäisiin tasaisin aikavälein. Online-tilassa olevia koneita voitaisiin seurata siten, että kirjaussovellus lähettäisi palvelimelle tasaisin aikavälein niin kutsutun heartbeat-pyynnön, jolla se ilmoittaa olevansa edelleen aktiivinen. Jos palvelin ei vastaanota uutta heartbeat-pyyntöä määritellyn ajan sisällä edellisestä, se päättelisi kirjaussovelluksen olevan offilne-tilassa.

Datansiirtomäärien ja resurssienkäytön tehokkuuden kannalta HTTP-pohjainen ratkaisu on kuitenkin selkeästi huonompi, kuin WebSocket- tai Server Sent Events -pohjaiset vaihtoehdot, sillä jokaisen pyynnön ja vastauksen mukana on HTTP-otsikkodataa ja useissa tilanteissa polling-pyynnöt saavat tyhjän vastauksen [19]. Näin ollen HTTP-pohjainen ratkaisuvaihtoehto hylätään ja vertaillaan tarkemmin WebSocket- ja Server Sent Events -ratkaisuja.

Sekä WebSocket-, että Server Sent Events -pohjainen ratkaisu soveltuisi hyvin kirjausjärjestelmän kehitystavoitteiden saavuttamiseen. Molemmat ratkaisut mahdollistavat yhteyden avaamisen ja datan lähettämisen palvelimelta kirjaussovellukselle käytännössä ilman viivettä. WebSocket-protokolla mahdollistaa myös datan lähettämisen saman yhteyden välityksellä asiakasohjelmalta palvelimelle, mikä ei kirjausjärjestelmän kehitystavoitteiden kannalta ole välttämätöntä, mutta jota voitaisiin käyttää päivitysten lähettämiseen kirjaussovellukselta palvelimelle. Saman yhteyden kautta voitaisiin myös seurata kirjaavan koneen online-tilaa. SSE-toteutuksella päivitykset lähetettäisiin kirjaussovellukselta palvelimelle normaalisti HTTP:n välityksellä ja online-tilaa seurattaisiin heartbeat-pyynnöillä.

Useimmissa verkkoselaimissa on olemassa valmis rajapinta sekä WebSocket-, että SSE:n EventSource -toteutuksille, joka hoitaa yhteyden teknisen hallinnan ja jonka kautta viestien vastaanottaminen on yksinkertaista. Ainoastaan Internet Explorer- ja

Edge -selaimista puuttuu tuki SSE:lle, mutta se ei olisi suuri haitta, sillä kirjausjärjestelmä on ensisijaisesti suunniteltu toimimaan ja käytettäväksi Google Chrome -selaimella. Myös Play Frameworkista löytyy valmis tuki molempien ratkaisujen toteutukselle. Tämä onnistuu Play Frameworkin Akka Actor-mallilla.

WebSocket-protokolla ja Server Sent Events ovat molemmat verkkoyhteyden sekä asiakas- ja palvelinohjelmien kuormituksen ja suoritustehokkuuden suhteen selkeästi parempia, kuin aiemmin käsitellyt HTTP polling-ratkaisut. Toisiinsa verrattuna niillä ei ole huomattavaa eroa. [19]

WebSocket- ja Server Sent Events- pohjaisten toteutusten välillä ei tämän tutkimuksen perusteella ole huomattavaa eroa tehokkuuden tai toteutettavuuden kannalta. Server Sent Events olisi riittävä ratkaisu, vaikka se tarjoaakin vähemmän ominaisuuksia siten, että se ei mahdollista kaksisuuntaista liikennettä. Kuitenkin ylläpidettävyyden ja jatkokehitettävyyden suhteen suoritekirjausjärjestelmään on päätetty ottaa käyttöön WebSocket-protokolla. Tämä mahdollistaa sen, että järjestelmään voidaan helposti tuoda tulevaisuudessa ominaisuuksia, jotka edellyttävät WebSocket-protokollan tarjoamia mahdollisuuksia.

5.2 Protokollan toteutus

WebSocket-yhteyksien hallinta toteutettiin palvelimen puolella Play Frameworkin Akka Actor-mallia hyödyntäen. Asiakasohjelmien puolella WebSocket tuki toteutettiin sekä raportointi-, että kirjaussovelluksille ja siihen käytettiin JavaScriptin WebSocket-rajapintaa. Palvelimen ja asiakasohjelmien kesken määriteltiin WebSocket-viestien rakenteelle yhtenäinen protokolla, jolle toteutettiin tuki molempiin päihin. Tässä luvussa käsitellään näiden toteutuksien yksityiskohtia.

5.2.1 Viestiprotokollan määrittely

Palvelimen ja asiakasohjelmien välinen WebSocket-viestintä toteutettiin pääasiassa JSON-formaattia noudattavien merkkijonojen avulla. JSON-formaatti valittiin, koska se on sekä ihmis-, että koneluettavuuden kannalta selkeä tapa lähettää rakenteista dataa. Lisäksi se on hyvin yleinen tapa datan lähettämiseen modernissa web-ympäristössä ja sen tulkinta on helposti toteutettavissa sekä Scalalla palvelimen puolella, että JavaScriptillä asiakasohjelmien päässä.

Protokollan suunnittelussa otettiin huomioon jatkokehitettävyys ja saman protokollan myöhempi sovellettavuus siten, että yksittäinen viesti käsittää tietynlaisen tapahtuman. Viesti koostuu siis ylimmällä tasollaan tapahtumakentästä ja datakentästä. Tämän työn

puitteissa ainoa tapahtuma on *machine-update* eli työkoneen tilapäivitys, mutta tämä menettely mahdollistaa helpommin jatkossa myös sellaisten viestien käsittelyn, joilla on erilainen rakenne.

Työkoneiden tilapäivitysten datakentälle määriteltiin rakenne, jota kaikki viestinnän vastapuolet noudattavat ja osaavat tulkita. Jokaisella tilapäivitysviestillä on viestityyppi *eventType*, joka kertoo, millainen tapahtuma on kyseessä. Määritellyt viestityypit ovat:

- *StartedWorkshift*: kone on aloittanut uuden työvuoron,
- *StartedProject*: kone on aloittanut uuden projektin nykyisen työvuoron sisällä,
- *StartedTuc*: kone on aloittanut työn uudella ajankäytön koodilla nykyisen projektin sisällä,
- *FinishedWorkshift*: kone on päättänyt työvuoron,
- *SuspendedWorkshift*: työvuoro on keskeytetty eli käytännössä kirjaussovellus on suljettu, kun kirjaus on ollut kesken,
- *ContinuedWorkshift*: keskeytettyä työvuoroa on jatkettu,
- *LostConnection*: verkkoyhteys koneen työvuoroa kirjaavaan kirjaussovellukseen on katkennut,
- *RegainedConnection*: katkennut verkkoyhteys on muodostettu uudelleen.

Erillisiä viestityyppejä ajokoodien tai projektien päättymiselle ei tarvita, sillä sellaiset tapahtumat voidaan päätellä muista viestityypeistä. Esimerkiksi ajokoodin tiedetään päättyneen, jos seuraavan päivityksen tyyppi on esimerkiksi uuden ajokoodin aloittava *StartedTuc* tai koko työvuoron päättävä *FinishedWorkshift*.

Viestityypin lisäksi tilapäivitykselle määriteltyjä kenttiä ovat:

- *user*: päivityksen lähettäneen eli kirjausta tekevän käyttäjän sähköpostiosoite.
- *machineId*: työkoneen tunniste.
- *workType*: päivityksen lähetyksenaikaisen ajokoodin numero. Jos päivityksen tapahtumahetkellä ei ole aktiivista ajokoodia, tämä on 0.
- *workSubType*: ajokoodin alatyypin numero, eli käytännössä tehollisen työn tapauksessa joko 1 tai 2, muussa tapauksessa 0.
- *projectNumber*: aktiivisen projektin projektinumero. Jos aktiivista projektia ei ole, tämä on 0.
- *phase*: aktiivisen projektin vaihe. Jos aktiivista projektia ei ole, tämä on 0.

- *task*: aktiivisen projektin tehtävänumero. Jos aktiivista projektia ei ole, tämä on 0.

Nämä kentät ovat mukana jokaisessa viestissä. Näin yksittäisestä viestistä voidaan tulkita informaatiota laajasti sen sijaan, että tiedettäisiin vain tapahtuman tyyppi. Esimerkiksi yhdestä *RegainedConnection*-tyypin viestistä voidaan tulkita sellainen tilanne, että kirjaavan laitteen verkkoyhteys on muodostettu uudelleen, mutta ajokoodi ei ole enää sama, kuin yhteyden katketessa, vaan se on muuttunut yhteyden ollessa poikki. Esimerkki tällaisesta viestistä on esitetty ohjelmassa 6.

```
{
  event: 'machine-update',
  data: {
    eventType: 'RegainedConnection',
    user: 'erkki@esimerkki.fi',
    machineId: '965f5e63-a462-4db0-a665-c68ee86d079d',
    workType: 1,
    workSubType: 2,
    projectNumber: 1234,
    phase: 55555,
    task: 66
  }
}
```

Ohjelma 6. *Esimerkki RegainedConnection-tyypin viestistä, jossa on mukana työtieto*

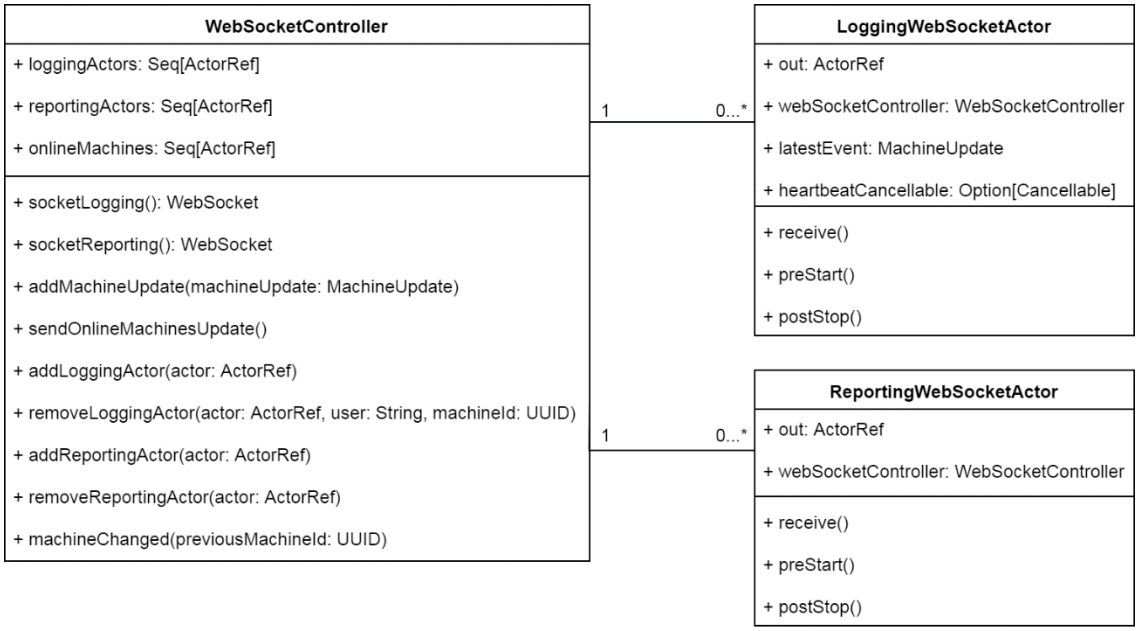
Tilapäivitysviestien lisäksi järjestelmään kehitettiin heartbeat-toiminto kirjaussovelluksen ja palvelimen välille. Alkuperäisen suunnitelman mukaan heartbeat-toiminnolle ei olisi ollut WebSocket-ratkaisun kanssa tarvetta, mutta kehitysvaiheessa huomattiin, että yhteyden katkeamisen automaattinen tunnistus ei ollut riittävän responsiivinen, vaan tieto siitä tuli vaihtelevalla ja ajoittain suurellakin viiveellä. Heartbeat-toiminto lisättiin siis tuomaan yhteyden tilan seurantaan luotettavuutta.

Heartbeat-toiminnon protokolla määriteltiin siten, että kirjaussovellus lähettää kymmenen sekunnin välein WebSocket-yhteyden välityksellä palvelimelle viestin ”heartbeat”. Kun palvelin vastaanottaa tämän viestin, se nollaa laskurin, jolla se seuraa kyseisen kirjaussovelluksen heartbeat-viestejä. Jos palvelin ei ole vastaanottanut heartbeat-viestiä kirjaussovellukselta 35 sekuntiin, eli jos viesti on jäänyt vastaanottamatta kolme kertaa, se päättelee yhteyden katkenneen ja lähettää raportointisovelluksille tästä kertovan tilapäivityksen.

5.2.2 Palvelin

WebSocket-tuki toteutettiin palvelimen puolella siten, että järjestelmään luotiin uusi Play Frameworkin kontrolleri, *WebSocketController*, jolle WebSocket-yhteyksien avauspyyn-

nöt ohjataan. Lisäksi raportointi- ja kirjaussovelluksien yhteyksiä varten luotiin Akka Actor-luokat *ReportingWebSocketActor* ja *LoggingWebSocketActor*. *WebSocketController* luo uusille yhteyksille Actor-oliot ja pitää listaa olemassa olevista raportointi- ja kirjausactoreista. Näiden listojen avulla se voi lähettää päivityksen esimerkiksi kaikille palvelimeen yhteydessä oleville raportointisovelluksille raportointiactorien kautta. Lisäksi *WebSocketController* ylläpitää tietoa online-tilassa olevista työkoneista säilyttämällä jokaisen koneen uusimman, online-tilaa indikoivan päivityksen listassa. Palvelimen WebSocket-toteutuksen luokkakaavio on esitetty kuvassa 7.



Kuva 7. *Palvelimen WebSocket-toteutuksen luokkakaavio*

WebSocketController, *ReportingWebSocketActor* ja *LoggingWebSocketActor* suunniteltiin vuorovaikuttamaan keskenään ja asiakassovellusten kanssa siten, että *ReportingWebSocketActor* ja *LoggingWebSocketActor* saavat viitteen *WebSocketControlleriin*. Näin *WebSocketController* toimii rajapintana, jonka metodeja *ReportingWebSocketActor* ja *LoggingWebSocketActor* voivat kutsua vastaanottaessaan esimerkiksi uusia päivityksiä asiakasohjelmalta. *WebSocketController* hoitaa kokonaisuuden käsittelyn ja voi tarvittaessa vastaavasti lähettää viestejä asiakasohjelmille Actor-olioiden välityksellä. Lisäksi se on yhteydessä järjestelmän muihin palveluihin.

Actor-olio luodaan esimerkiksi raportointisovelluksen WebSocket-yhteydelle ohjelman 7 mukaisesti. WebSocket-yhteyden avaamispyyntö ohjataan *WebSocketControllerin socketReporting()*-metodille, joka tarkistaa, onko asiakassovellus autentikoitunut ja jos on, se luo uuden *ReportingWebSocketActor*-luokan, antaa sille viitteen itsestään ja avaa yhteyden. Jos asiakassovellus ei ole autentikoitunut, se ei avaa yhteyttä ja palauttaa HTTP-vastauksen statuskoodilla *403 Forbidden*. Kun Actor-olio syntyy, sen *preStart()*-

metodi ajetaan. Tässä tapauksessa *ReportingWebSocketActor* kutsuu *preStart()*-metodissaan *WebSocketControllerin addReportingActor()*-metodia antaen parametrina viitteen itseensä, jolloin *WebSocketController* lisää *ReportingWebSocketActorin reportingActors*-listaansa. *LoggingWebSocketActor* luodaan ja yhdistetään asiakassovelluksen vastaavasti.

```
def socketReporting = WebSocket.acceptOrResult[String, String] { request =>
  Future.successful(request.session.get("uuid") match {
    case Some(uuid) => {
      Right(ActorFlow.actorRef { out =>
        ReportingWebSocketActor.props(out, this)
      })
    }
    case None => {
      Left(Forbidden)
    }
  })
}
```

Ohjelma 7. *Akka Actorin luominen raportointisovelluksen WebSocket-yhteydelle*

ReportingWebSocketActor on toteutukseltaan yksinkertainen. Sen tehtävä on välittää *WebSocketControllerilta* sille tulevat viestit sille asiakasohjelmalle, jonka yhteyttä se hallitsee. Kun Actor-oliolle lähetetään muuttuja, sen *receive()*-metodi ottaa vastaan ja käsittelee sen. *ReportingWebSocketActorin receive()*-metodi määriteltiin vastaanottamaan ja käsittelemään muuttujia, joiden tyyppi on *MachineUpdate* eli työkoneen tilapäivitys tai *OnlineMachinesMessage* eli lista online-tilassa olevien työkoneiden uusimmista tilapäivityksistä. Kumman tahansa vastaanottaessaan se muuttaa viestin JSON-formaattiin ja edelleen merkkijonoksi ja lähettää merkkijonon WebSocket-yhteyden välityksellä asiakasohjelmalle. Kun WebSocket-yhteys katkeaa esimerkiksi asiakasohjelman sulkeutuessa, ajetaan Actor-olion *postStop()*-metodi ja olio tuhotaan. *ReportingWebSocketActorin postStop()*-metodi kutsuu *WebSocketControllerin removeReportingActor()*-metodia, jolloin *WebSocketController* poistaa *ReportingWebSocketActorin reportingActors*-listastaan.

LoggingWebSocketActor on yhteydessä kirjaussovellukseen ja sen tehtävä on vastaanottaa ja käsitellä kirjaussovellukselta tulevat työkoneen tilapäivitykset ja informoida niiden perusteella *WebSocketControlleria* tilapäivityksistä ja muista tapahtumista. Lisäksi se sisältää heartbeat-mekanismin, jolla se pitää kirjaa yhteydestään kirjaussovellukseen.

LoggingWebSocketActorin receive()-metodi vastaanottaa tekstimuotoisia viestejä kirjaussovellukselta. Sen pääasiallisesti vastaanottamat viestit ovat koneen tilapäivityksiä JSON-muotoisina merkkijonoina. Se tulkitsee viestit *MachineUpdate*-muotoon ja lähet-

tää päivityksen eteenpäin *WebSocketControllerille* kutsumalla sen *addMachineUpdate()*-metodia. Tämän jälkeen *WebSocketController* tallentaa päivityksen tietokantaan, lähettää päivityksen *ReportingWebSocketActoreille* ja kokoaa ja lähettää *ReportingWebSocketActoreille* uuden listan online-tilassa olevista koneista. Jos kirjaussovelluksen käyttäjä on valinnut kirjaukselleen uuden koneen ja päivityksessä on eri kone, kuin *LoggingWebSocketActorin* säilyttämässä viimeisimmässä päivityksessä, se ilmoittaa siitä *WebSocketControllerille* kutsumalla sen *machineChanged()*-metodia, jolloin *WebSocketController* poistaa vanhan koneen online-tilassa olevien koneiden listalta.

5.2.3 Asiakasohjelmat

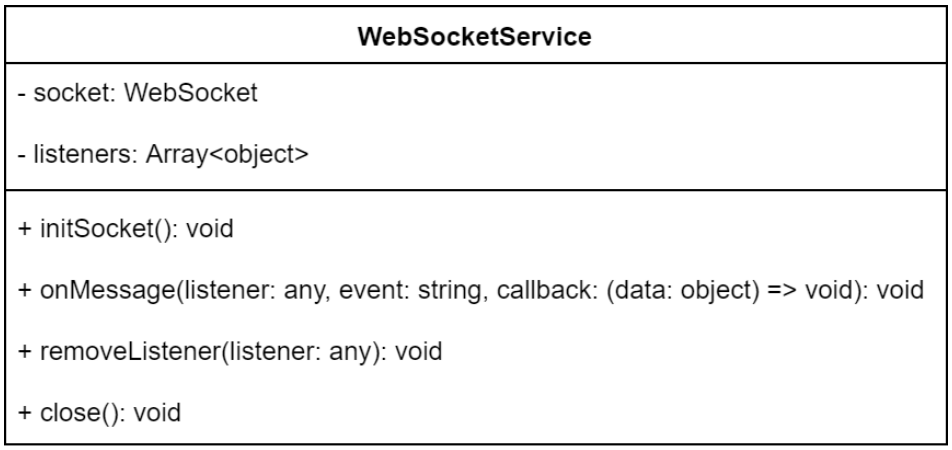
Raportointi- ja kirjaussovelluksille WebSocket-tuki toteutettiin JavaScriptin WebSocket-rajapinnan avulla. Rajapinta on suhteellisen yksinkertainen ja sen kirjausjärjestelmän kannalta tärkeimmät ominaisuudet ovat yhteyden tilan tarkistamiseen käytettävä *readyState*-ominaisuus, sen erilaiset tiettyjen tapahtumien aikaan ajettavat funktiot, kuten *onopen*, *onmessage* ja *onclose* sekä sen metodit *send()* ja *close()* [20].

WebSocket-luokan *readyState*-ominaisuuden mahdolliset tilat ovat:

- *WebSocket.CONNECTING (0)*,
- *WebSocket.OPEN (1),*
- *WebSocket.CLOSING (2),*
- *WebSocket.CLOSED (3)* [20].

Näitä hyödynnetään esimerkiksi kirjaussovelluksessa yhteyden tilan tarkkailuun ja tarvittaessa yhteyden uudelleen muodostamiseen.

Kirjausjärjestelmän kirjaus- ja raportointisovelluksille luotiin uusi Angular-palvelu *WebSocketService*, joka hoitaa yhteyden avaamisen, sulkemisen, viestien lähetyksen ja kuuntelun ja mahdolliset uudelleenyhdistämisyritykset. Ne tarjoavat sovellusten muille palveluille ja komponenteille rajapinnan esimerkiksi yhteyden avaamiseen ja viestien lähettämiseen. Raportointisovelluksen *WebSocketServicen* luokkakaavio on esitetty kuvassa 8.



Kuva 8. *Raportointisovelluksen WebSocketServicen luokkakaavio*

Raportointisovelluksen *WebSocketServicen socket*-muuttuja on JavaScriptin *WebSocket*-objekti, joka hoitaa yhteyden tekniset yksityiskohdat. WebSocket-yhteys avataan luomalla uusi WebSocket-objekti antaen sen parametriksi palvelimen WebSocket-yhteyden avaamiseen tarkoitetun URL-osoitteen [15]. Metodi *initSocket()* avaa WebSocket-yhteyden palvelimeen ja määrittelee, mitä tapahtuu, kun WebSocket-viesti vastaanotetaan määrittämällä funktioarvon *socket.onmessage*-muuttujalle. Tämä tapahtuu ohjelman 8 mukaisesti.

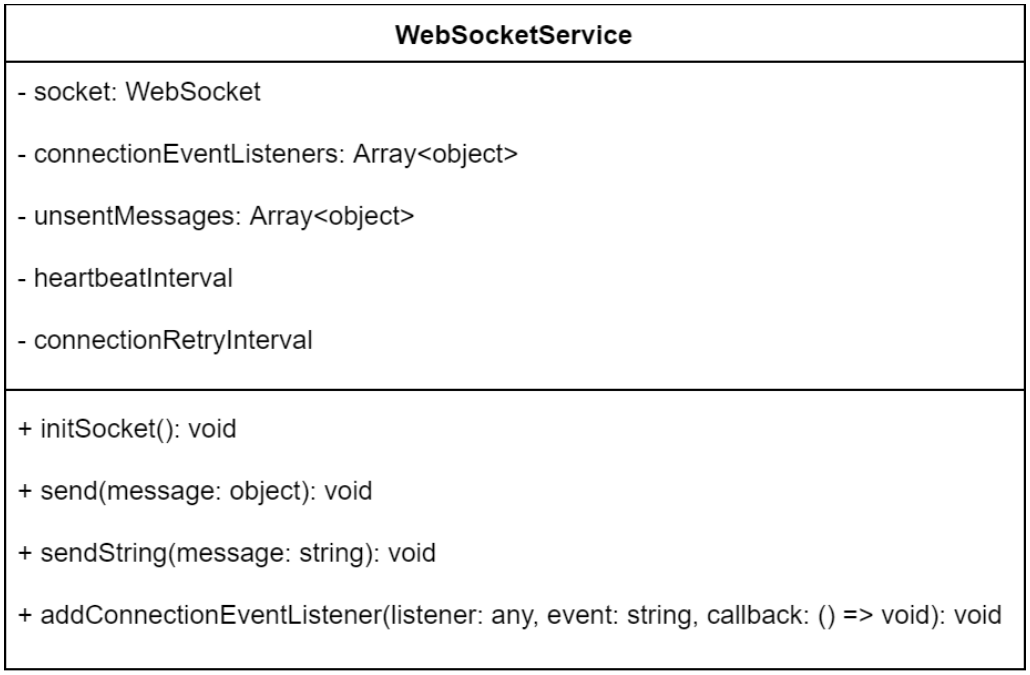
```
this.socket.onmessage = (event) => {
  console.log('WebSocket received message:', event.data);
  try {
    const eventJSON: {event: string, data: any} = JSON.parse(event.data);
    self.listeners.forEach(listener => {
      if (listener.event === eventJSON.event) {
        listener.callback(eventJSON);
      }
    });
  } catch (SyntaxError) {
    console.log('Received message is not JSON');
  }
};
```

Ohjelma 8. *Raportointisovelluksen toiminta, kun WebSocket-viesti vastaanotetaan*

Kun WebSocket-viesti vastaanotetaan, sen oletetaan olevan JSON-formaattia noudattava merkkijono, joten se käännetään JavaScript-objektiksi *JSON.parse()*-funktion avulla. Tämän jälkeen käydään läpi *WebSocketServicelle* rekisteröidyt kuuntelijat, ja jos jokin niistä kuuntelee viestissä ilmoitettua tapahtumatyyppiä, viesti välitetään sille. Raportointisovelluksen muut komponentit voivat rekisteröidä kuuntelijoita tapahtumatyypeille *WebSocketServicen onMessage()*-metodin avulla.

Raportointisovelluksen *WebSocketServicellä* on myös *close()*-metodi WebSocket-yhteyden sulkemista varten. WebSocket-yhteyttä tarvitaan vain tähän työhön liittyvässä seurantanäkymässä, joten yhteys voidaan sulkea, kun näkymästä poistutaan.

Kirjaussovelluksen *WebSocketServicen* toteutus on hieman monipuolisempi suunnitellun protokollan erilaisten vaatimusten vuoksi. Kirjaussovelluksen *WebSocketServicen* luokkakaavio on esitetty kuvassa 9.



Kuva 9. *Kirjaussovelluksen WebSocketServicen luokkakaavio*

Kirjaussovelluksen puolella *WebSocketServicen initSocket()*-metodi avaa WebSocket-yhteyden palvelimeen, kuten raportointisovelluksessa ja lisäksi se määrittää toimintaa *WebSocket*-objektin *onopen*- ja *onclose*-tapahtumille. Kun WebSocket-yhteys luodaan, *WebSocketService* lähettää heartbeat-viestin palvelimelle ja asettaa intervallimuuttujan *heartbeatInterval* toistamaan heartbeat-viestiä kymmenen sekunnin välein. Jos se havaitsee, että avautunut yhteys on aiemmin katkennut yhteys ja sillä on palvelimelle lähettämättömiä viestejä muuttujassa *unsentMessages*, se lähettää ne palvelimelle. Kun yhteys sulkeutuu, se lopettaa heartbeat-viestien lähettämisen nollaamalla muuttujan *heartbeatInterval* ja aloittaa yhteyden uudelleenmuodostusyrityksen asettamalla intervallimuuttujan *connectionRetryInterval* yrittämään yhteyden muodostusta kymmenen sekunnin välein.

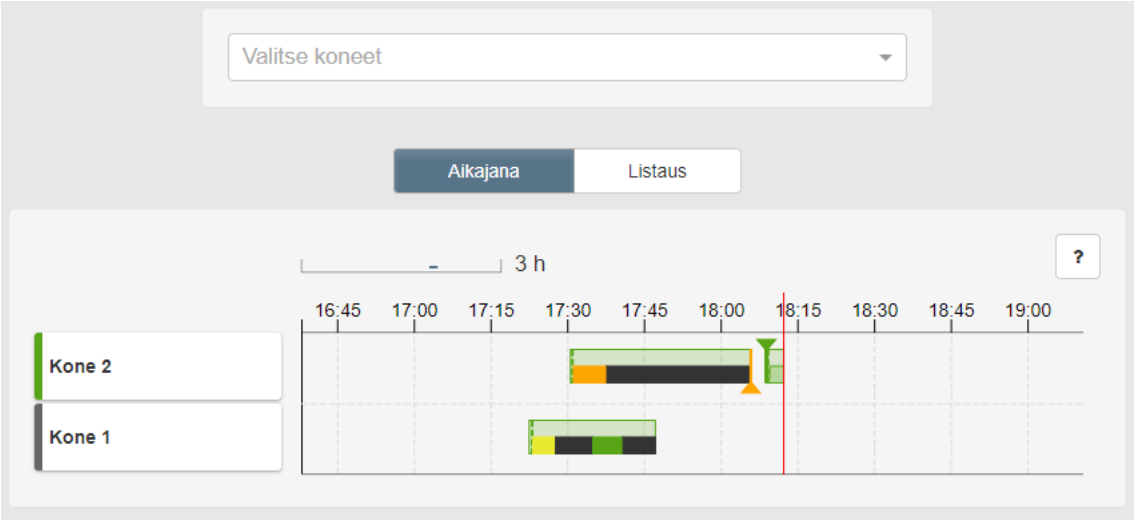
Kirjaussovelluksen *WebSocketServicen send()*-metodilla sovelluksen muut komponentit voivat lähettää palvelimelle JavaScript-objekteja JSON-formaattia noudattavina merkkijonoina eli sen avulla sovellus lähettää palvelimelle työkoneiden tilapäivityksiä. Tilapäi-

vityksiä lähetetään eri komponenteista samoissa tilanteissa, joissa niitä kirjataan työvuorolle myös vanhan toiminnan mukaan. Metodi *sendString()* on pelkän merkkijonon lähettämistä varten ja sitä käytetään heartbeat-viestien lähettämiseen. Metodin *addConnectionEventListener()* avulla muut komponentit voivat pyytää tiedon yhteyden tilan muutoksista, jolloin *WebSocketService* ilmoittaa niistä niiden tapahtuessa.

5.3 Käyttöliittymä

Käyttöliittymän osalta järjestelmään haluttiin työkoneiden seurantasivu. Tämä sivu suunniteltiin jakautumaan aikajana- ja listausnäkymiin. Aikajananäkymään suunniteltiin gantt-kaavio, jossa on nähtävillä työkoneet ja niiden toiminta viimeisen 72 tunnin ajalta. Listausnäkymään suunniteltiin kronologinen listauskomponentti, jossa on nähtävillä koneiden viimeisimmät tilapäivitykset 72 tunnin ajalta uusimmasta vanhimpaan. Molemmissa näkymissä näytettäviä koneita voidaan rajata näkymän yläreunassa olevalla suodatusvalinnalla.

Aikajananäkymä näyttää gantt-kaavion, jossa on lueteltu riveittäin työkoneet, jotka ovat lähettäneet tilapäivityksiä viimeisen 72 tunnin aikana. Ylimmäisenä on lueteltu tällä hetkellä aktiivisesti kirjausta tekevät koneet ja ne on ilmaistu vihreällä korostusvärillä koneen nimen vieressä. Offline-tilassa olevat koneet on korostettu harmaalla värillä. Aikajananäkymän ulkoasu on esitetty kuvassa 10.



Kuva 10. *Seurantasivun aikajananäkymä*

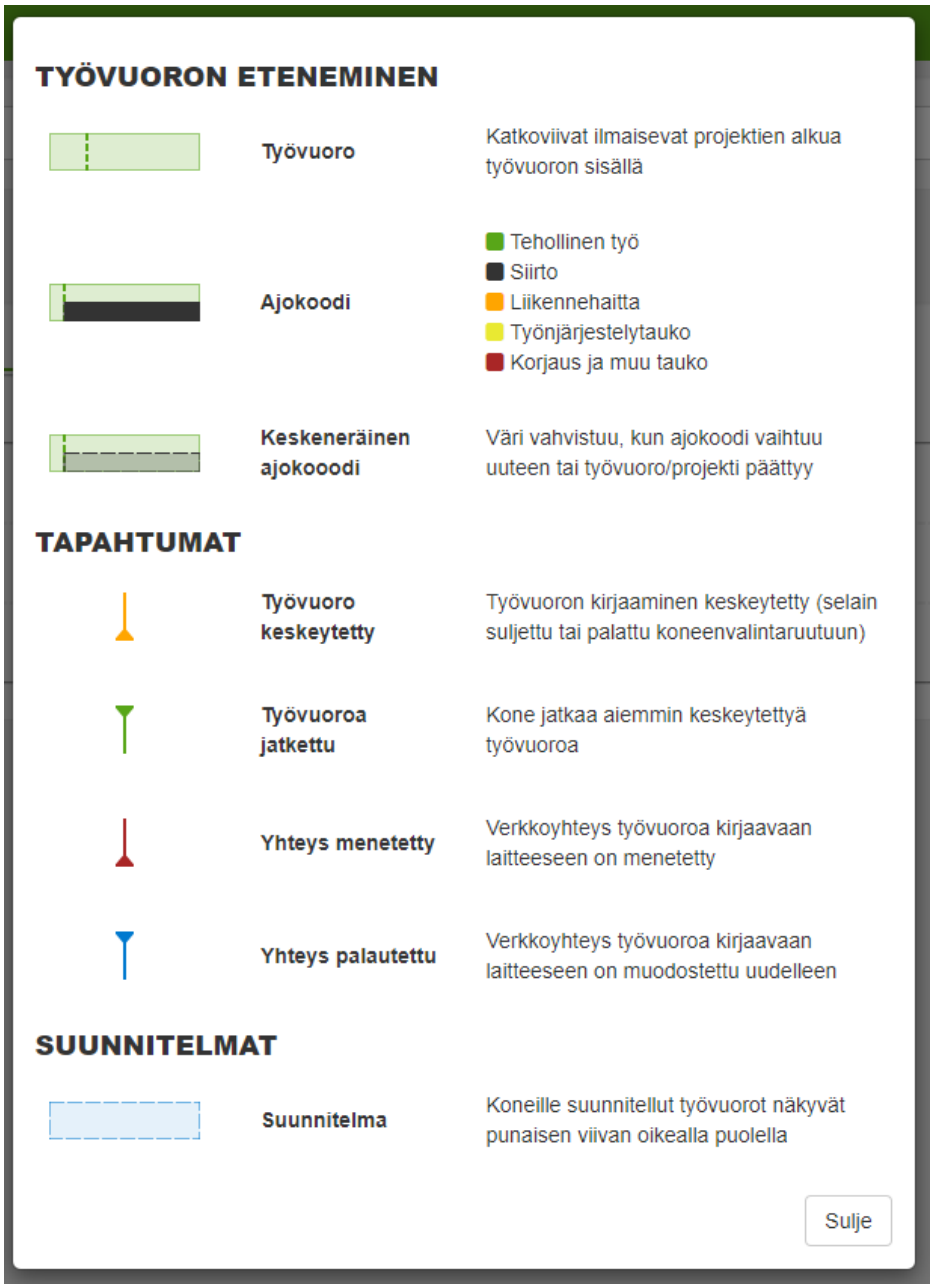
Aikajanan vaaka-akselilla on nähtävissä päivämäärät ja kellonajat. Aikajana on skaalattu siten, että siinä näkyy 72 tuntia menneisyyteen ja kahdeksan tuntia tulevaisuuteen. Nykyhetki on korostettu punaisella viivalla. Aikajana ulottuu tulevaisuuteen, koska siinä näytetään myös koneille suunnitellut työvuorot. Näin aikajanalta voidaan tarkkailla suunniteltujen työvuorojen toteutumista.

Aikajanan skaalausta voidaan suurentaa ja pienentää hiiren rullalla ja aikajanalla voidaan liikkua vaakasuunnassa hiirellä vetämällä. Aikajanan yläkulmassa on visuaalinen indikaattori, joka kuvaa nykyisen näkymän sijaintia ja suurennuksen tasoa aikajanalla.

Aikajanan toisessa yläkulmassa on painike, josta avautuu aikajanan merkintöjen opastekomponentti. Aikajanalla työvuorot on kuvattu haaleana vihreänä palkkina ja niiden sisässä on esitetty projektit, ajokoodit ja muut tapahtumat. Kun hiiri viedään työvuoropalkin päälle, tooltip-viesti näyttää työvuoron alku- ja loppukellonajat. Projektien alut on esitetty pystysuorana vihreänä katkoviivana työvuoropalkissa. Ajokoodit on esitetty pienempänä palkkina työvuoropalkin sisällä ja ne on värikoodattu. Keskeneräiset ajokoodit esitetään haalealla värillä ja valmiit kiinteällä värillä. Ajokoodien tooltip-viestissä näkyy ajokoodin nimi ja sen alku- ja loppukellonajat. Työvuorosuunnitelmat on esitetty haaleana sinisenä palkkina.

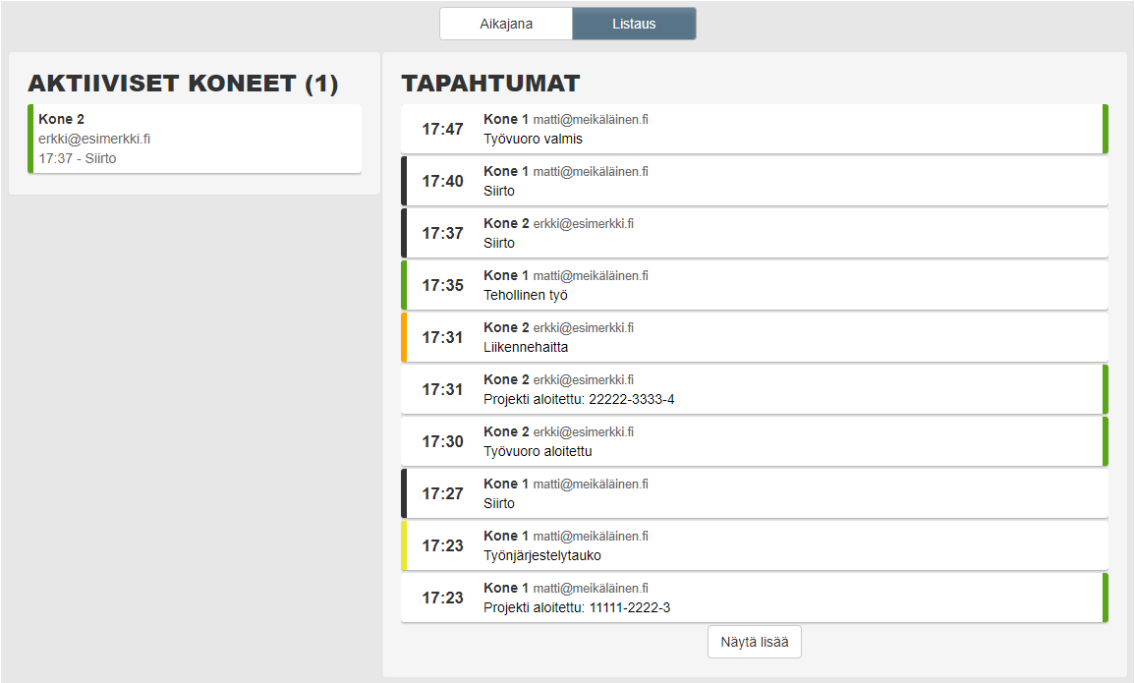
Erityistapahtumat on kuvattu työvuoropalkin päälle tulevana pystysuorana merkintänä. Merkinnät on värikoodattu ja ne ovat tapahtumasta riippuen hieman eri muotoisia. Merkinnän tooltip-viestissä näkyy sen tarkoitus ja tapahtuman kellonaika. Aikajanan visuaaliset elementit on tarkemmin esitetty niiden opastuskomponenttia esittävässä kuvassa 11.

Aikajananäkymän grafiikka toteutettiin D3.js -kirjaston [21] avulla. D3.js on JavaScript-kirjasto datan visualisointiin. Sen avulla visualisointi pystyttiin toteuttamaan dynaamisesti DOM:iin svg-elementin sisään. Kaavion päivittäminen toteutettiin siten, että se piirretään uudelleen kymmenen sekunnin välein, jolloin työvuoropalkkeja piirretään pidemmäksi samalla tilanteella. Kaavio piirretään uudestaan myös silloin, kun vastaanotetaan uusi koneen tilapäivitys. Kun kaaviota suurennetaan tai siirretään hiirellä, sitä piirretään uudelleen jatkuvasti monitorin virkistystaajuuden mukaan.



Kuva 11. *Aikajananäkymän merkintöjen opastuskomponentti*

Listausnäkymässä näkyy vasemmalla sivulla lista aktiivisista koneista, niiden kuljettajista sekä tämänhetkisestä toiminnasta. Pääkomponenttina on lista tapahtumista. Kun uusi tapahtuma vastaanotetaan, se lisätään automaattisesti listaan. Listausnäkymän ulkoasu on esitetty kuvassa 12.



Kuva 12. *Seurantasivun listausnäkymä*

Listausnäkymässä tapahtumilla on ajokoodikohtainen värikoodein toteutettu korostus vasemmassa sivussa. Yhteyden tilaa indikoivat päivitykset on korostettu vihrein ja punaisin värikoodein oikeassa sivussa.

6. TOTEUTUKSEN ARVIOINTI

Tämän työn ensisijainen toiminnallinen kehitystavoite oli tehdä järjestelmän kirjausprosessin informaationkulusta reaaliaikaisempaa, jotta esimiehet voivat seurata työvuorojen kulkua myös niiden tapahtuessa, eikä vasta, kun työvuoro on valmis ja lähetetty palvelimelle. Valitulla tiedonkulun toteutustavalla ja toteutetulla raportointisovelluksen käyttöliittymällä tämä tavoite on saavutettu.

Toteutettujen kaavio- ja listausnäkymien avulla esimies voi reaaliaikaisesti seurata työvuoroja, saaden välitöntä tietoa niiden alkamisesta ja päättymisestä, projektien ja ajokoodien vaihtumisesta, potentiaalista ongelmatilanteista, työvuoron keskeytymisestä ja jatkumisesta sekä yhteyden katkeamisesta ja uudelleen muodostumisesta. Lisäksi kaavionäkymästä nähdään konekohtaisesti tulevien työvuorosuunnitelmien ajankohdat, joten niiden toteutumista voidaan myös seurata.

Työllä oli kaksi teknistä tavoitetta, joista ensisijainen oli löytää ja toteuttaa reaaliaikaiselle kaksisuuntaiselle kommunikaatiolle ratkaisumenetelmä, joka on verkkoyhteyden ja laitteiston kuormituksen suhteen tehokas, joka on järkevästi toteutettavissa olemassa olevaan järjestelmään ja joka potentiaalisesti tarjoaisi samaa teknologiaa hyödyntäen toteutettavissa olevia jatkokehitysmahdollisuuksia. Tämän lisäksi seurantakäyttöliittymän suunnittelussa ja toteutuksessa tavoiteltiin hyvää käytettävyyttä ja informatiivisuutta.

Reaaliaikaisen seurannan toteutus kaksisuuntaisen kommunikaatiomenetelmän avulla onnistui suurimmilta osin sen mukaisesti, kuten vaihtoehtoisten toteutustapojen tutkimisen jälkeen oli suunniteltu. Alkuperäisen suunnitelman mukaan heartbeat-mekanismia kirjaussovelluksen ja palvelimen välillä ei olisi tarvittu, mutta WebSocket-yhteyden katkeamisen automaattisen tunnistamisen epäluotettavan viiveen vuoksi se päätettiin ottaa mukaan toteutukseen. Tähän liittyen päätös käyttää WebSocket-yhteyttä HTTP:n sijaan myös kirjaussovelluksen ja palvelimen välillä oli hyödyllinen, sillä säännöllisten heartbeat-viestien välittäminen WebSocket-yhteyden kautta kuormittaa verkkoyhteyttä HTTP:ta vähemmän.

Toteutustapoja vertaillessa todettiin, että WebSocket- ja Server Sent Events -pohjaiset ratkaisut olisivat toteutettavuuden ja resurssienkäytön suhteen hyvin samanlaisia. Tätä samanlaisuutta korostaa entisestään todettu tarve heartbeat-mekanismille myös WebSocket-pohjaisessa ratkaisussa. Server Sent Events -ratkaisun kanssa heartbeat-mekanismi olisi kuitenkin toteutettu HTTP:n välityksellä, joten WebSocket-ratkaisu to-

dennäköisesti säilyy resurssienkäytön suhteen hieman tehokkaampana. Kuitenkin, kuten vertailussa arvioitiin, Server Sent Events -ratkaisu olisi myös ollut riittävä järjestelmän kehitystavoitteiden kannalta ja WebSocket-ratkaisuun päädyttiin sen avaamien jatkokehitysmahdollisuuksien vuoksi.

Toteutettavuutensa puolesta WebSocket-ratkaisu oli hyvä. Järjestelmässä olemassa olevista teknologiaratkaisuista löytyi valmiita kirjastoja WebSocket-yhteyksien käyttöönottoon. Toteutus onnistui enimmäkseen hyvin suoraviivaisesti käytettyjen teknologioiden dokumentaatiota ja ohjeita tutkien ja noudattaen. Eniten omaa suunnittelua tarvittiin yhteyksien käsittelyyn halutulla tavalla, heartbeat-mekanismin toteutukseen sekä viestiprotokollan määrittelyyn ja viestien protokollanmukaiseen käsittelyyn.

Sekä kaavio-, että listausnäkymä onnistuttiin toteuttamaan tavoitteiden mukaisesti. Kaavionäkymään toteutettiin halutun kaltainen gantt-kaavio, joka päivittyy reaaliajassa ja jota voidaan suurentaa, pienentää ja siirtää hiiren avulla helpottaen sen luettavuutta ja siten parantaen sen käytettävyyttä. Kaavionäkymään toteutettiin myös opastuskomponentti, josta kaaviossa käytettyjen merkintöjen tarkoitukset käyvät ilmi. Suunniteltujen merkintöjen avulla kaavio on työvuorojen yksityiskohtien suhteen informatiivinen, mutta samalla kompakti. Listausnäkymään toteutettiin tapahtumalistaus toivotulla tavalla. Kuten toteutustavoitteissa määriteltiin, molempia näkymiä voidaan tarvittaessa rajata valitsemalla konevalikosta koneet, joita halutaan seurata.

Reaaliaikaisen viestinnän toteutuksessa keskeisenä haasteena nähtiin työkoneiden kirjauksessa käytettävä mobiiliverkkoyhteys ja sen mahdollinen epävakaus riippuen sijainnista, jossa työtä ja kirjausta tehdään. Tämä haluttiin ottaa huomioon toteutusmenetelmän valinnassa, toteutuksen suunnittelussa sekä itse toteutuksessa. Tavoitteena oli, että yhteyksien hallinta toimii oikein ajoittain katkeavista yhteyksistä huolimatta ja että seurantakäyttöliittymä olisi katkenneiden yhteyksien suhteen riittävän informatiivinen.

Yhteyksien hallinta onnistuttiin Play Frameworkin Actor-mallin, suunniteltujen viestityyppien ja heartbeat-mekanismin avulla toteuttamaan siten, että katkeavat yhteydet eivät aiheuta ongelmia tai sekoita kirjausten seurantaa millään tavalla. Yhteyden katketessa heartbeat-viestien vastaanottaminen loppuu, jonka perusteella viesti yhteyden katkeamisesta voidaan toimittaa raportointisovelluksille ja tieto siitä näyttää käyttöliittymässä tapahtumamerkintänä. Myös kirjaussovellus tietää yhteyden katkenneen, joten sen palatessa se lähettää yhteyden uudelleenmuodostuksesta ilmoittavan viestin palvelimelle, joka toimitetaan raportointisovelluksille ja näytetään seurantakäyttöliittymässä uutena ta-

pahtumamerkintänä. Näin ollen palvelimen ei tarvitse säilyttää tietoa siitä, että jokin yhteys on katkennut, vaan riittää, että se välittää tiedon yhteyden uudelleenmuodostuksesta eteenpäin.

Reaaliaikaisen seurannan tuotantokäyttöönoton jälkeen seurantakäyttöliittymän kaavionäkymään liittyen on huomattu potentiaalinen käytettävyyden parannuskohde. Seurantakäyttöliittymä pysyy raportointisovelluksen vastaanottamien tapahtumien avulla informatiivisena yhteyksien katkeamisen suhteen, mutta on havaittu, että joissain tilanteissa huonon mobiiliverkkoyhteyden alueella toimivan koneen yhteys katkeilee ja muodostuu uudelleen jatkuvasti. Kun kaikki nämä yhteyden katkeamiset ja uudelleenmuodostumiset näytetään aikajanalla omina merkintöinään, käyttöliittymä voi näyttää ruuhkaiselta. Tilannetta voitaisiin parantaa suunnittelemalla käyttöliittymään mekanismi, jonka avulla tiheään ilmaantuvien merkintöjen näkyvyyttä esimerkiksi rajoitettaisiin tai ne esitettäisiin eri tavalla. Tämä seikka ei kuitenkaan ole erittäin merkittävä kokonaisuuden kannalta, joten toimintaa sen suhteen ei ole päätetty lähteä kehittämään.

Kokonaisuutena työkoneiden reaaliaikaisen seurannan toteutus voidaan katsoa onnistuneeksi. Kehitystavoitteet toteutettavien ominaisuuksien ja teknisten vaatimusten suhteen on täytetty, vaikka paikoin tutkimuksen perusteella tehdystä suunnitelmasta jouduttiin toteutusvaiheessa poikkeamaan.

7. JATKOKEHITYSMAHDOLLISUUDET

Reaaliaikaisen kaksisuuntaisen kommunikaation toteutustavaksi valittiin WebSocket-pohjainen ratkaisu osittain sen tarjoamien jatkokehitettävyysmahdollisuuksien vuoksi. Jatkokehitettävyys otettiin myös huomioon toteutuksen suunnittelussa. Backend-järjestelmään luotiin WebSocket-yhteyksiä hallitseva kontrolleri, jota voidaan tarvittaessa helposti laajentaa uusilla WebSocket-protokollaa hyödyntävillä ominaisuuksilla ja erilaisille protokollan käyttötavoille voidaan luoda omat Actor-luokat.

Kirjaus- ja raportointisovelluksille luotiin Angularin palvelut, jotka hoitavat WebSocket-yhteyden käsittelyn ja tarjoavat siihen liittyviä toimintoja muille komponenteille. Uusia ominaisuuksia toteutettaessa näitä toimintoja voidaan hyödyntää ja tarvittaessa palvelua voidaan myös helposti laajentaa uusilla ominaisuuksilla.

Viestiprotokolla suunniteltiin siten, että uusien viestityyppien suunnittelu ja järjestelmään lisääminen on helppoa. Uusia ominaisuuksia toteutettaessa täytyy vain toteuttaa uusien viestityyppien käsittely backend-palvelimella ja kirjaus- ja raportointisovellusten komponenteissa.

Eräs potentiaalinen reaaliaikaiseen seurantaan läheisesti liittyvä jatkokehitysvaihtoehto on reaaliaikaisten viestien lähettäminen raportointisovelluksen ja kirjaussovelluksen välillä. Tällaisella mekanismilla meneillään olevista työvuoroista voitaisiin saada esimiehille entistä monipuolisempaa informaatiota. Tämä voisi olla hyödyllistä esimerkiksi työvuoron aikana kohdattujen vikatilanteiden pelkkää ajokoodia tarkemmassa kuvaamisessa ja mahdollisten toimintaohjeiden välittämisessä. Lisäksi esimies voisi tarvittaessa lähettää työkoneen miehistölle informaatiota mahdollisista työvuoroon vaikuttavista odottamattomista tekijöistä. Tällaisen ominaisuuden toteuttaminen onnistuisi helposti olemassa olevalla WebSocket-ratkaisulla, sillä WebSocket-yhteys otettiin toteutusvaiheessa käyttöön myös kirjaussovelluksen ja palvelimen välillä.

Valmiin WebSocket-tuen pohjatoteutusta voidaan hyödyntää myös yleisesti raportointisovelluksessa. Esimerkiksi työvuorojen hyväksyntänäkymässä WebSocket-protokollan mahdollistamaa palvelimelta asiakasohjelmalle suuntautuvaa liikennettä voitaisiin hyödyntää työvoroille tapahtuvien muutosten reaaliaikaiseen päivittymiseen. Teoriassa nykyisellä toteutuksella on esimerkiksi mahdollista, että kaksi esimiestä ovat samalla ajan hetkellä hyväksymässä, hylkäämässä tai muokkaamassa tiettyä työvuoroa, jolloin jälkimmäinen muutos korvaisi ensimmäisen ja olisi ainut, joka jäisi voimaan. Jos ensimmäi-

sen muutoksen tekijän tekemä muutos tulisi näkyviin myös toisen esimiehen käyttöliittymässä reaaliaikaisesti ilman sivun päivittämistä, tällaista tilannetta ei pääsisi tapahtumaan.

Toinen WebSocket-protokollaa hyödyntävä raportointisovelluksen yleisen tason jatkokehitysmahdollisuus on ilmoitusominaisuus. Raportointisovellus voisi jatkuvasti pitää avointa WebSocket-yhteyttä palvelimeen, jolloin palvelin voisi lähettää sille ilmoituksia erilaisista tapahtumista. Tällaisia tapahtumia voisivat olla esimerkiksi uusi aloitettu työvuoro, työvuoron valmistuminen tai kirjaussovelluksesta lähetetty uusi kuva, tankkausilmoitus tai viesti. Ilmoitukset voitaisiin esittää raportointisovelluksen käyttöliittymässä esimerkiksi ruudun oikeaan yläkulmaan ilmestyvässä elementissä. Myös kirjaussovelluksen puolella voitaisiin näyttää ilmoituksia esimerkiksi esimiesten lähettämistä viesteistä tai muista tiedotteista.

8. YHTEENVETO

Tämän työn tavoite oli reaaliaikaistaa ratatyökoneiden suoritekirjausjärjestelmän työvuoronkirjausprosessin tiedonkulkua. Järjestelmään luotiin seurantakäyttöliittymä, jonka avulla työvuoroja ja niille kirjattuja tapahtumia voidaan seurata reaaliaikaisesti sen sijaan, että työvuoroista saataisiin tieto vasta, kun se on valmis ja lähetetty kokonaisuudessaan palvelimelle.

Reaaliaikaisen seurannan toteutus vaati menetelmän, jolla tietoa voidaan lähettää reaaliaikaisesti verkkopalvelimelta asiakasohjelmalle. Käytettävän menetelmän vaatimuksiksi asetettiin tehokkuus sen datankäytön suhteen, hyvä toteutettavuus olemassa olevaan järjestelmään ja hyvät kehitettävän pohjatoteutuksen tarjoamat jatkokehitysmahdollisuudet. Toteutustapojen vertailu tehtiin HTTP-protokollan soveltamisen, WebSocket-protokollan ja Server Sent Events -teknologian välillä. Vertailun perusteella toteutustavaksi valittiin WebSocket-protokolla, koska sen katsottiin täyttävän parhaiten edellä mainitut vaatimukset.

Reaaliaikaiselle viestinnälle määriteltiin viestiprotokolla, jolle toteutettiin tuki järjestelmän palvelimella ja asiakasohjelmissa. Tuki toteutettiin myös itse WebSocket-protokollalle. Tätä toteutusta hyödyntämään rakennettiin seurantakäyttöliittymä, jossa työvuorojen etenemistä voidaan seurata yksityiskohtaisesti ja reaaliajassa kaavionäkymässä tai tapahtumalistauksena.

Valmiilla toteutuksella saavutettiin työn kehitystavoitteet ja toteutuksen voidaan katsoa onnistuneen. WebSocket-protokolla oli hyvä valinta kaksisuuntaisen viestinnän pohjatoteutukselle ja sille onnistuttiin suunnittelemaan erilaisia jatkokehitysvaihtoehtoja.

LÄHTEET

[1] Enrico -juhlapyhärajapinta. Saatavissa (viitattu 19.10.2018): https://kayaposoft.com/enrico/docs-v1.0/json/

[2] Liikenneviraston Infra-API -rajapinta. Saatavissa (viitattu 19.10.2018): https://rata.digitraffic.fi/infra-api/0.2/swagger-ui.html

[3] Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Saatavissa (viitattu 21.4.2019): https://tools.ietf.org/html/rfc7230

[4] D. Barker, The OSI Network Model Explained, Database & Network Journal, 2012, Vol. 42 Issue 5, p3-4, 2p

[5] Transmission Control Protocol. Saatavissa (viitattu 21.4.2019): https://tools.ietf.org/html/rfc793

[6] An Overview of HTTP. Saatavissa (viitattu 13.2.2019): https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[7] Play Framework. Saatavissa (viitattu 22.4.2019): https://www.playframework.com/

[8] D. Pop, A. Altar, Designing an MVC Model for Rapid Web Application Development, Procedia Engineering, Volume 69, 2014, pp 1172-1179

[9] Akka Documentation. Saatavissa (viitattu 22.4.2019): https://doc.akka.io/docs/akka/current/index.html

[10] A. Freeman, Pro Angular 6, Third Edition, Apress, 2018

[11] S. Fenton, Pro TypeScript: Application-Scale JavaScript Development, Second Edition, Apress, 2018

[12] J. Wallace, JSON Quick Syntax Reference, Apress, 2016

[13] Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. Saatavissa (viitattu 13.2.2019): https://tools.ietf.org/html/rfc6202

[14] The WebSocket Protocol. Saatavissa (viitattu 13.2.2019): https://tools.ietf.org/html/rfc6455

[15] Writing WebSocket client applications. Saatavissa (viitattu 19.3.2019): https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_client_applications

[16] Play Framwork Documentation, WebSockets. Saatavissa (viitattu 19.3.2019): https://www.playframework.com/documentation/2.5.x/ScalaWebSockets

[17] J. Reid, HTML5 Programmer's Reference, Apress, 2015, 350 p.

[18] Instant Messaging at LinkedIn: Scaling to Hundreds of Thousands of Persistent Connections on One Machine. Saatavissa (viitattu 19.3.2019): https://engineering.linkedin.com/blog/2016/10/instant-messaging-at-linkedin--scaling-to-hundreds-of-thousands-

[19] A Comparison Between WebSockets, Server-Sent Events, And Polling. Saatavissa (viitattu 17.3.2019): https://aquil.io/articles/a-comparison-between-websockets-server-sent-events-and-polling

[20] WebSocket. Saatavissa (viitattu 31.3.2019): https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

[21] D3.js. Saatavissa (viitattu 31.3.2019): https://d3js.org/